

libc_yacc.h

```
#define ST_TSO 768
#define ST_TSOI 769
#define ST_TSE 770
#define ST_TSEI 771
#define ST_TSC 772
#define ST_TSCA 773
#define ST_TSCB 774
#define ST_TCLK 775
#define REV_V 776
#define L_DEF_REAL 777
#define L_DEF_TEXT 778
#define L_DEF_BOOL 779
#define L_INT 780
#define L_REAL 781
#define L_STRING 782
#define L_NSTRING 783
#define L_ESTRING 784
#define L_QSTRING 785
#define SDF_SKIP 786
```

```
extern YYSTYPE libclval;
```

libc_yacc.y

```
%{

/*=====
=====

#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stdarg.h>
#include <malloc.h>

/* ----- */

void libcerror(char *S);
void lex_prev_state(void);
void lex_next_pin_state(void);

#include "libc_def.h"

extern char *yytext;
extern int scaling_attr;

FILE *libcin;           /* yyin */

char *file_name;

#define KF()      libc_cell_k_factor()

%}

%token K_LIBRARY /* library */
%token K_ANPPP      /* aux_no_pulldown_pin_property */
%token K_BNS        /* bus_naming_style */
%token K_COMMENT    /* comment */
%token K_CU         /* current_unit */
%token K_DATE       /* date */
%token K_DM         /* delay_model */
%token K_IPSM       /* in_place_swap_mode */
%token K_LPU        /* leakage_power_unit */
%token K_MWE        /* max_wired_emitters */
%token K_MDL        /* multiple_drivers_legal */
%token K_NP         /* nom_process */
%token K_NT         /* nom_temperature */
%token K_NV         /* nom_voltage */
%token K_NTIDF     /* nonpaired_twin_inc_delay_func */
%token K_NPPP       /* no_pulldown_pin_property */
%token K_PT         /* piece_type */
%token K_PU         /* power_unit */
%token K_POPSRC    /* preferred_output_pad_slew_rate_control */
%token K_POPV       /* preferred_output_pad_voltage */
%token K_PIPV       /* preferred_input_pad_voltage */
%token K_PRU        /* pulling_resistance_unit */
%token K_RC         /* reference_capacitance */
%token K_REVISION   /* revision */
%token K_SIMULATION /* simulation */
```

libc_yacc.y

```
%token K_TU          /* time_unit */
%token K_UPP         /* unconnected_pin_property */
%token K_VU          /* voltage_unit */
%token K_WLF         /* wired_logic_function */

%token K_DCLP        /* default_cell_leakage_power */
%token K_DCP          /* default_cell_power */
%token K_DCC          /* default_connection_class */
%token K_DFO          /* default_edge_rate_breakpoint_f0 */
%token K_DF1          /* default_edge_rate_breakpoint_f1 */
%token K_DR0          /* default_edge_rate_breakpoint_r0 */
%token K_DR1          /* default_edge_rate_breakpoint_r1 */
%token K_DEC          /* default_emitter_count */
%token K_DFDI         /* default_fall_delay_intercept */
%token K_DFNT         /* default_fall_nonpaired_twin */
%token K_DFPR         /* default_fall_pin_resistance */
%token K_DFWR         /* default_fall_wire_resistance */
%token K_DFWE         /* default_fall_wor_emmitter */
%token K_DFWI         /* default_fall_wor_intercept */
%token K_DFL          /* default_fanout_load */
%token K_DIOPC        /* default_inout_pin_cap */
%token K_DIOFR        /* default_inout_pin_fall_res */
%token K_DIIRR        /* default_inout_pin_rise_res */
%token K_DIPC          /* default_input_pin_cap */
%token K_DIF           /* default_intrinsic_fall */
%token K_DIR           /* default_intrinsic_rise */
%token K_DMC          /* default_max_capacitance */
%token K_DMFO         /* default_max_fanout */
%token K_DMT          /* default_max_transition */
%token K_DMU          /* default_max_utilization */
%token K_DMP          /* default_min_porosity */
%token K_DOC           /* default_operating_conditions */
%token K_DOPC          /* default_output_pin_cap */
%token K_DOFR          /* default_output_pin_fall_res */
%token K_DORR          /* default_output_pin_rise_res */
%token K_DPL           /* default_pin_limit */
%token K_DPP           /* default_pin_power */
%token K_DRFC          /* default_rc_fall_coefficient */      /* cmos2 */
%token K_DRRC          /* default_rc_rise_coefficient */      /* cmos2 */
%token K_DRDI          /* default_rise_delay_intercept */
%token K_DRNT          /* default_rise_nonpaired_twin */
%token K_DRPR          /* default_rise_pin_resistance */
%token K_DRWR          /* default_rise_wire_resistance */
%token K_DRWE          /* default_rise_wor_emitter */
%token K_DRWI          /* default_rise_wor_intercept */
%token K_DSC           /* default_setup_coefficient */      /* */
%token K_DSF           /* default_slope_fall */
%token K_DSR           /* default_slope_rise */
%token K_DWL           /* default_wire_load */
%token K_DWLA          /* default_wire_load_area */
%token K_DWLC          /* default_wire_load_capacitance */
%token K_DWLM          /* default_wire_load_mode */
%token K_DWLR          /* default_wire_load_resistance */
%token K_DWLS          /* default_wire_load_selection */

%token P_CR           /* k_process_cell_rise (temp/volt) */
```

libc_yacc.y

```
%token P_CF
%token P_CLP
%token P_CP
%token P_DC
%token P_DF
%token P_DR
%token P_FDI
%token P_FPR
%token P_FP
%token P_FT
%token P_FWR
%token P_FWE
%token P_FWI
%token P_HF
%token P_HR
%token P_IP
%token P_IF
%token P_IR
%token P_MP
%token P_MPWH
%token P_MPWL
%token P_NF
%token P_NR
%token P_PC
%token P_PP
%token P_RF
%token P_RR
%token P_REF
%token P_RER
%token P_RDI
%token P_RPR
%token P_RP
%token P_RT
%token P_RWR
%token P_RWE
%token P_RWI
%token P_SF
%token P_SR
%token P_SKF
%token P_SKR
%token P_SLF
%token P_SLR
%token P_WC
%token P_WR

%token K_CLU
%token K_DEFINE
%token K_DCA
%token K_LF
%token K_PD
%token K_RL
%token K_TECH

%token K_CELL
%token K_IV
%token K_LTT
%token K_OC

/* k_process_cell_fall (temp/volt) */
/* k_process_cell_leakage_power (temp/volt) */
/* k_process_cell_power (temp/volt) */
/* k_process_drive_current (temp/volt) (old) */
/* k_process_drive_fall (temp/volt) (old) */
/* k_process_drive_rise (temp/volt) (old) */
/* k_process_fall_delay_intercept (temp/volt) */
/* k_process_fall_pin_resistance (temp/volt) */
/* k_process_fall_propagation (temp/volt) */
/* k_process_fall_transition (temp/volt) */
/* k_process_fall_wire_resistance (temp/volt) */
/* k_process_fall_wor_emitter (temp/volt) */
/* k_process_fall_wor_intercept (temp/volt) */
/* k_process_hold_fall (temp/volt) */
/* k_process_hold_rise (temp/volt) */
/* k_process_internal_power (temp/volt) */
/* k_process_intrinsic_fall (temp/volt) */
/* k_process_intrinsic_rise (temp/volt) */
/* k_process_min_period (temp/volt) */
/* k_process_min_pulse_width_high (temp/volt) */
/* k_process_min_pulse_width_low (temp/volt) */
/* k_process_nochange_fall (temp/volt) */
/* k_process_nochange_rise (temp/volt) */
/* k_process_pin_cap (temp/volt) */
/* k_process_pin_power (temp/volt) */
/* k_process_recovery_fall (temp/volt) */
/* k_process_recovery_rise (temp/volt) */
/* k_process_removal_fall (temp/volt) */
/* k_process_removal_rise (temp/volt) */
/* k_process_rise_delay_intercept (temp/volt) */
/* k_process_rise_pin_resistance (temp/volt) */
/* k_process_rise_propagation (temp/volt) */
/* k_process_rise_transition (temp/volt) */
/* k_process_rise_wire_resistance (temp/volt) */
/* k_process_rise_wor_emitter (temp/volt) */
/* k_process_rise_wor_intercept (temp/volt) */
/* k_process_setup_fall (temp/volt) */
/* k_process_setup_rise (temp/volt) */
/* k_process_skew_fall (temp/volt) */
/* k_process_skew_rise (temp/volt) */
/* k_process_slope_fall (temp/volt) (old) */
/* k_process_slope_rise (temp/volt) (old) */
/* k_process_wire_cap (temp/volt) */
/* k_process_wire_res (temp/volt) */

/* capacitive_load_unit */
/* define */
/* define_cell_area */
/* library_features */
/* piece_define */
/* routing_layers */
/* technology */

/* cell */
/* input_voltage */
/* lu_table_template */
/* operating_conditions */
```

libc_yacc.y

```
%token K_OV          /* output_voltage */
%token K_PCELL       /* parameterized_cell */
%token K_PLT         /* power_lut_template */
%token K_PS          /* power_supply */
%token K_RTD         /* rise_transition_degradation */
%token K_FTD         /* fall_transition_degradation */
%token K_SC          /* scaled_cell */
%token K_SF          /* scaled_factors */
%token K_TR          /* timing_range */
%token K_TYPE        /* type */
%token K_WL          /* wire_load */
%token K_WLS         /* wire_load_selection */
%token K_WLT         /* wire_load_table */

%token AREA          /* area */
%token C_APC         /* auxiliary_pad_cell */
%token C_CF          /* cell_footprint */
%token C_POWER        /* cell_power */
%token C_LP          /* cell_leakage_power */
%token C_CC          /* contention_condition */
%token C_DF          /* dont_false */
%token C_DT          /* dont_touch */
%token C_GP          /* geometry_print */
%token C_DU          /* dont_use */
%token C_HNC         /* handle_negative_constraint */
%token C_IT          /* interface_timing (old) */
%token C_MO          /* map_only */
%token C_PC          /* pad_cell */
%token C_PT          /* pad_type */
%token C_PL          /* pin_limit */
%token C_P           /* preferred */
%token C_SF          /* scaling_factors */
%token C_SG          /* scan_group */
%token C_SBD         /* single_bit_degenerate */
%token C_VN          /* vhdl_name */

%token C_PE          /* pin_equal */
%token C_PO          /* pin_opposite */
%token C_RC          /* rail_connection */

%token C_BUNDLE       /* bundle */
%token C_BUS          /* bus */
%token C_FF           /* ff */
%token C_FFB          /* ff_bank */
%token C_IP           /* internal_power */
%token C_LPG          /* leakage_power */
%token C_LATCH         /* latch */
%token C_LB           /* latch_bank */
%token C_LUT          /* lut */          /* for FPGA */
%token C_MEM          /* memory */
%token C_PIN          /* pin */
%token C_RT           /* routing_track */
%token C_STATE         /* state */
%token C_ST           /* statetable */
%token C_TC           /* test_cell */

%token MEMBERS        /* members */
```

libc_yacc.y

```
%token BUS_TYPE          /* bus_type */
%token MEM_READ          /* memory_read */
%token MEM_WRITE /* memory_write */

%token CE_PROPERTY        /* cell_property */
%token CE_DE              /* default_enum */
%token CE_PP              /* parameterized_pin */
%token PP_PROPERTIES      /* pin_properties */
%token PP_DISABLE /* disabled */

%token CLOCK_ON           /* clocked_on */
%token NEXT_ST            /* next_state */
%token CLEAR              /* clear */
%token PRESET             /* preset */
%token CL_PS_V1           /* clear_preset_var1 */
%token CL_PS_V2           /* clear_preset_var2 */
%token ON_ALSO            /* clock_on_also, enable_on_also */
%token ENABLE              /* enable */
%token DATA_IN             /* data_in */
%token FORCE_01            /* force_01 */
%token FORCE_10            /* force_10 */
%token FORCE_00            /* force_00 */
%token FORCE_11            /* force_11 */

%token REL_INP             /* related_input */
%token REL_INPS            /* related_inputs */
%token REL_OUTP            /* related_output */
%token VALUES              /* values */
%token IP_E000             /* equal_or_opposite_output */
%token IP_PL               /* power_level */
%token IP_FP               /* fall_power */
%token IP_RP               /* rise_power */
%token IP_POWER            /* power */
%token VALUE               /* value */

%token LUT_IP              /* input_pins */

%token CM_TYPE             /* type (in memory group) */
%token CM_RAM              /* ram */
%token CM_ROM              /* rom */
%token CM_ADDR_WIDTH       /* address_width */
%token CM_WORD_WIDTH       /* word_width */
%token CM_C_ADDR /* column_address */
%token CM_R_ADDR /* row_address */

%token PIN_CAP              /* capacitance */
%token PIN_CLK              /* clock */
%token PIN_CGEP             /* clock_gate_enable_pin */
%token PIN_CC               /* connection_class */
%token PIN_DIR              /* direction */
%token PIN_DF               /* dont_false */
%token PIN_DC               /* drive_current */
%token PIN_DT               /* driver_type */
%token PIN_ERBF0 /* edge_rate_breakpoint_f0 */      /* cmos2 */
%token PIN_ERBF1 /* edge_rate_breakpoint_f1 */      /* cmos2 */
%token PIN_ERBR0 /* edge_rate_breakpoint_r0 */      /* cmos2 */
%token PIN_ERBR1 /* edge_rate_breakpoint_r1 */      /* cmos2 */
```

```
%token PIN_ERF          /* edge_rate_fall */           /* cmos2 */
%token PIN_ERR          /* edge_rate_rise */          /* cmos2 */
%token PIN_ERLF         /* edge_rate_load_fall */        /* cmos2 */
%token PIN_ERLR         /* edge_rate_load_rise */       /* cmos2 */
%token PIN_EC           /* emitter_count */           /* cmos2 */
%token PIN_FCSAT        /* fall_current_slop_after_threshold */ /* */
%token PIN_FCSBT        /* fall_current_slop_before_threshold */ /* */
%token PIN_FTAT          /* fall_time_after_threshold */      /* */
%token PIN_FTBT          /* fall_time_before_threshold */     /* */
%token PIN_FWE           /* fall_wor_emitter */           /* */
%token PIN_FWI           /* fall_wor_intercept */          /* */
%token PIN_FL            /* fanout_load */             /* */
%token PIN_FUNCTION      /* function */               /* */
%token PIN_H             /* hysteresis */              /* */
%token PIN_IM            /* input_map */              /* */
%token PIN_ISL           /* input_signal_level */        /* */
%token PIN_IV            /* input_voltage */            /* */
%token PIN_IN            /* internal_node */           /* */
%token PIN_IO            /* inverted_output */          /* */
%token PIN_IP            /* is_pad */                 /* */
%token PIN_MAX_FO        /* max_fanout */              /* */
%token PIN_MAX_TRANS     /* max_transition */          /* */
%token PIN_MAX_CAP       /* max_capacitance */         /* */
%token PIN_MIN_FO        /* min_fanout */              /* */
%token PIN_MIN_TRANS     /* min_transition */          /* */
%token PIN_MIN_CAP       /* min_capacitance */         /* */
%token PIN_MP            /* min_period */              /* */
%token PIN_MPWH          /* min_pulse_width_high */     /* */
%token PIN_MPWL          /* min_pulse_width_low */       /* */
%token PIN_MPP           /* multicell_pad_pin */        /* */
%token PIN_MDL           /* multiple_drivers_legal */    /* */
%token PIN_NST           /* next_state_type */          /* */
%token PIN_OSL           /* output_signal_level */        /* */
%token PIN_OV            /* output_voltage */           /* */
%token PIN_PFT           /* pin_func_type */           /* */
%token PIN_PP            /* pin_power */              /* */
%token PIN_PT            /* prefer_tied */              /* */
%token PIN_PO            /* primary_output */           /* */
%token PIN_PC            /* pulling_current */          /* */
%token PIN_PR            /* pulling_resistance */        /* */
%token PIN_RC            /* reference_capacitance */     /* */
%token PIN_RCSAT         /* rise_current_slop_after_threshold */ /* */
%token PIN_RCSBT         /* rise_current_slop_before_threshold */ /* */
%token PIN_RTAT          /* rise_time_after_threshold */    /* */
%token PIN_RTBT          /* rise_time_before_threshold */   /* */
%token PIN_RWE           /* rise_wor_emitter */          /* */
%token PIN_RWI           /* rise_wor_intercept */         /* */
%token PIN_SC            /* slew_control */             /* */
%token PIN_SF            /* state_function */           /* */
%token PIN_TS            /* three_state */              /* */
%token PIN_VN            /* vhdl_name */              /* */
%token PIN_WC            /* wire_capacitance */          /* */
%token PIN_WCC           /* wired_connection_class */     /* */
%token PIN_XF            /* x_function */              /* */
%token TIMING           /* timing */                 /* */
%token MIN_PLUSE_WIDTH   /* min_pulse_width */          /* */
%token MIN_PERIOD         /* minimum_period */           /* */
```

libc_yacc.y

```
%token PIN_IPO          /* internal_power */

%token ADDRESS           /* address */
%token TRACKS            /* tracks */
%token TRACK_AREA /* total_track_area */
%token TABLE             /* table */

%token CTC_PIN            /* pin in test_cell() */
%token CTC_DIR             /* direction */
%token CTC_FUNC            /* function */
%token CTC_ST              /* signal_type */
%token CTC_TOO             /* test_output_only */

%token TI_ERSF0           /* edge_rate_sensitivity_f0 */
%token TI_ERSF1           /* edge_rate_sensitivity_f1 */
%token TI_ERSR0           /* edge_rate_sensitivity_r0 */
%token TI_ERSR1           /* edge_rate_sensitivity_r1 */
%token TI_FR               /* fall_resistance */
%token TI_RR               /* rise_resistance */
%token TI_IF               /* intrinsic_fall */
%token TI_IR               /* intrinsic_rise */
%token TI_RBP              /* related_bus_pins */
%token TI_ROP              /* related_output_pin */
%token TI_RP               /* related_pin */
%token TI_SDF_C            /* sdf_cond */
%token TI_SDF_CS           /* sdf_cond_start */
%token TI_SDF_CE           /* sdf_cond_end */
%token TI_SDF_E            /* sdf_edges */
%token TI_SF               /* slope_fall */
%token TI_SR               /* slope_rise */
%token TI_TT               /* timing_type */
%token TI_TS               /* timing_sense */
%token TI_WHEN             /* when */
%token TI_WS               /* when_start */
%token TI_WE               /* when_end */

%token TI_FDI              /* fall_delay_intercept */
%token TI_FNT              /* fall_nonpaired_twin */
%token TI_FPR              /* fall_pin_resistance */
%token TI_FWR              /* fall_wire_resistance */
%token TI_RDI              /* rise_delay_intercept */
%token TI_RNT              /* rise_nonpaired_twin */
%token TI_RPR              /* rise_pin_resistance */
%token TI_RWR              /* rise_wire_resistance */

%token CELL_DEGR           /* cell_rise */
%token CELL_FALL            /* cell_fall */
%token CELL_RISE           /* cell_rise */
%token R_PROP              /* rise_propagation */
%token F_PROP              /* fall_propagation */
%token R_TRANS             /* rise_transition */
%token F_TRANS             /* fall_transition */
%token R_CONS              /* rise_constraint */
%token F_CONS              /* fall_constraint */

%token NO_EDGE              /* noedge */
%token BOTH_EDGES /* both_edges */
%token START_EDGE /* start_edge */
```

```

%token END_EDGE          /* end_edge */

%token MPW_CH            /* constraint_high */
%token MPW_CL            /* constraint_low */
%token MP_C              /* constraint */

%token IV_L               /* vil */
%token IV_H               /* vih */
%token IV_MIN             /* vimin */
%token IV_MAX             /* vimax */

%token TBL_VAR1           /* variable_1 */
%token TBL_VAR2           /* variable_2 */
%token TBL_VAR3           /* variable_3 */
%token TBL_IDX1           /* index_1 */
%token TBL_IDX2           /* index_2 */
%token TBL_IDX3           /* index_3 */
%token LTT_INT             /* input_net_transition / input_transition_time */
%token LTT_TONC            /* total_output_net_capacitance */
%token LTT_ONL             /* output_net_length */
%token LTT_ONWC            /* output_net_wire_cap */
%token LTT_ONPC            /* output_net_pin_cap */

%token LTT_ROTONC          /* related_out_total_output_net_capacitance */
%token LTT_ROONL            /* related_out_output_net_length */
%token LTT_ROONWC           /* related_out_output_net_wire_cap */
%token LTT_ROONPC           /* related_out_output_net_pin_cap */
%token LTT_CPT              /* constrained_pin_transition */
%token LTT_RPT              /* related_pin_transition */

%token LTT_OPT              /* output_pin_transition */
%token LTT_CD                /* connect_delay */

%token OC_PROCESS          /* process */
%token OC_TEMP              /* temperature */
%token OC_TREE              /* tree_type */
%token OC_VOLT              /* voltage */
%token OC_PR                /* power_rail */

%token OV_L                 /* vol */
%token OV_H                 /* voh */
%token OV_MIN               /* vomin */
%token OV_MAX               /* vomax */

%token PS_DPR              /* default_power_rail */

%token PC_CELL_ENUM         /* cell_enum */
%token TR_FF                /* faster_factor */
%token TR_SF                /* slower_factor */

%token TYPE_BASE            /* base_type */
%token TYPE_ARRAY           /* array */
%token TYPE_FROM             /* bit_from */
%token TYPE_TO               /* bit_to */
%token TYPE_WIDTH            /* bit_width */
%token TYPE_DT               /* data_type */
%token TYPE_BIT              /* bit */

```

libc_yacc.y

```
%token TYPE_DWNTO      /* dwnto */

%token WL_RES          /* resistance */
%token WL_SLOPE        /* slope */
%token WL_FL           /* fanout_length */
%token WLS_WLFA        /* wire_load_from_area */
%token WLT_AREA        /* fanout_area */
%token WLT_CAP         /* fanout_capacitance */
%token WLT_RES         /* fanout_resistance */

%token BOOL_T           /* true */
%token BOOL_F           /* false */

%token CU_A             /* A */
%token CU_MA            /* mA */
%token CU_UA            /* uA */

%token LU_MF            /* mf */
%token LU_UF            /* uf */
%token LU_NF            /* nf */
%token LU_PF            /* pf */
%token LU_FF            /* ff */

%token VU_V              /* V */
%token VU_MV            /* mV */

%token PU_W              /* W */
%token PU_MW            /* mW */
%token PU_UW            /* uW */
%token PU_NW            /* nW */
%token PU_PW            /* pW */

%token RU_OHM           /* ohm */
%token RU_KOHM          /* kohm */

%token TU_NS             /* ns */
%token TU_PS             /* ps */

%token DM_G_ECL          /* generic_ecl */
%token DM_G_CMOS         /* generic_cmos */
%token DM_TBL_LOOKUP      /* table_lookup */
%token DM_CMOS2          /* cmos2 */
%token DM_P_COMS         /* piecewise_cmos */

%token IPO_MF            /* match_footprint */
%token IPO_NS            /* no_swapping */
%token IPO_IF            /* ignore_footprint */

%token MMP_MAX           /* max */
%token MMP_MIN           /* min */
%token MMP_PLUS          /* plus */

%token PT_LENGTH          /* piece_length */
%token PT_WIRE_CAP        /* piece_wire_cap */
%token PT_PIN_CAP         /* piece_pin_cap */
%token PT_TOTAL_CAP       /* piece_total_cap */
```

libc_yacc.y

```
%token WLF_WAND      /* wired_and */
%token WLF_WOR       /* wired_or */

%token WLM_T          /* top */
%token WLM_S          /* segmented */
%token WLM_E          /* enclosed */

%token RT_PS          /* pad_slots */
%token RT_PDS         /* pad_driver_sites */
%token RT_PIDS        /* pad_input_driver_sites */
%token RT_PODS        /* pad_output_driver_sites */

%token TT_BEST        /* best_case_tree */
%token TT_BAL         /* balanced_tree */
%token TT_WORST        /* worst_case_tree */

%token SA_SA0          /* sa0 */
%token SA_SA1          /* sa1 */
%token SA_SA01         /* sa01 */

%token DIR_INPUT      /* input */
%token DIR_OUTPUT     /* output */
%token DIR_INOUT      /* inout */
%token DIR_INTERNAL   /* internal */

%token DRT_PULL_UP    /* pull_up */
%token DRT_PULL_DOWN  /* pull_down */
%token DRT_OPEN_DRAIN  /* pull_open_drain */
%token DRT_OPEN_SOURCE /* pull_open_source */
%token DRT_BUS_HOLD   /* pull_bus_hold */
%token DRT_RES         /* resistive */
%token DRT_RES0        /* resistive_0 */
%token DRT_RES1        /* resistive_1 */

%token N_DATA          /* data */
%token N_PRESET        /* preset */
%token N_CLEAR         /* clear */
%token N_LOAD          /* load */
%token N_SCAN_IN       /* scan_in */
%token N_SCAN_ENABLE   /* scan_enable */

%token CLK_ENABLE      /* clock_enable */
%token ACT_HIGH        /* active_high */
%token ACT_LOW         /* active_low */
%token ACT_RISING      /* active_rising */
%token ACT_FALLING    /* active_falling */

%token NONE_SC         /* none */
%token LOW_SC          /* low */
%token MED_SC          /* medium */
%token HIGH_SC         /* high */

%token TIT_RE          /* rising_edge */
%token TIT_FE          /* falling_edge */
%token TIT_PS          /* preset */
%token TIT_CL          /* clear */
%token TIT_HR          /* hold_rising */
```

libc_yacc.y

```
%token TIT_HF          /* hold_falling */
%token TIT_SR          /* setup_rising */
%token TIT_SF          /* setup_falling */
%token TIT_RR          /* recovery_rising */
%token TIT_RF          /* recovery_falling */
%token TIT_TSD         /* three_state_disable */
%token TIT_TSE         /* three_state_enable */
%token TIT_RMR         /* removal_rising */
%token TIT_RMF         /* removal_falling */
%token TIT_C           /* combinational */
%token TIT_SKR         /* skew_rising */
%token TIT_SKF         /* skew_falling */
%token TIT_NSHR        /* non_seq_hold_rising */
%token TIT_NSHF        /* non_seq_hold_falling */
%token TIT_NSSR        /* non_seq_setup_rising */
%token TIT_NSSF        /* non_seq_setup_falling */
%token TIT_NCHH        /* nochange_high_high */
%token TIT_NCHL        /* nochange_high_low */
%token TIT_NCLH        /* nochange_low_high */
%token TIT_NCLL        /* nochange_low_low */

%token TIS_POS          /* positive_unate */
%token TIS_NEG          /* negative_unate */
%token TIS_NON          /* non_unate */

%token ST_TSI           /* test_scan_in */
%token ST_TSII          /* test_scan_in_inverted */
%token ST_TSO            /* test_scan_out */
%token ST_TSOI           /* test_scan_out_inverted */
%token ST_TSE            /* test_scan_enable */
%token ST_TSEI           /* test_scan_enable_inverted */
%token ST_TSC            /* test_scan_clock */
%token ST_TSCA           /* test_scan_clock_a */
%token ST_TSCB           /* test_scan_clock_b */
%token ST_TCLK           /* test_clock */

%token REV_V             /* [0-9.] + ([a-zA-Z])? revision */

%token L_DEF_REAL /* {string} for define() attr */
%token L_DEF_TEXT /* {string} for define() attr */
%token L_DEF_BOOL /* {string} for define() attr */

%token L_INT              /* {integer} */
%token L_REAL             /* {real} */
%token L_STRING           /* {string} */
%token L_NSTRING          /* {nstring} for \"1A\" */
%token L_ESTRING           /* "10x10" */
%token L_QSTRING           /* {qstring} for statetable */

%token SDF_SKIP           /* for sdf expression */

%left '^'
%left '| '+' '
%left '&' '**'

%%
```

libc_yacc.y

```
/* ===== */
source_code      : { libc_def_init(); }
                  def_attrs. library_group
;

def_attrs.       : def_attrs
|
;

def_attrs        : def_attr
| def_attrs def_attr
;

def_attr         : string '=' n_expression semicolon.          /* define
value */
                  { libc_def_gc_insert($1.string,$3.real_val); }

/* ----- */
identifier       : L_STRING
| L_NSTRING
| L_ESTRING
;

/* ---- id or "id" */
string           : identifier
| ''' identifier '''
{ $$ = $2; }

QSTR_string      : L_QSTRING
| string    { lex_prev_state(); $$ = $1; }

name             : string
;

name.            : name
| { $$ .string = NULL; }

name_list        : name
{ libc_util_name_list1(&($$),&($1)); }
| name_list comma. name
{ libc_util_name_list2(&($$),&($1),&($3)); }

qname_list       : identifier
{ libc_util_name_list1(&($$),&($1)); }
| ''' id_list '''
{ $$ = $2; }

id_list          : identifier
{ libc_util_name_list1(&($$),&($1)); }
| id_list comma. identifier
```

libc_yacc.y

```
{ libc_util_name_list2(&($$), &($1), &($3)); }
```

```
;
```

```
real : sign L_INT { $$.real_val = $1.real_val * $2.int_val; }
```

```
| sign L_REAL { $$.real_val = $1.real_val * $2.real_val; }
```

```
;
```

```
sign : '+' { $$.real_val = 1.0; }
```

```
| '-' { $$.real_val = -1.0; }
```

```
| ';' { $$.real_val = 1.0; }
```

```
;
```

```
value : real
```

```
| ''' real ''' { $$.real_val = $2.real_val; }
```

```
| '{' real '}' { $$.real_val = $2.real_val; }
```

```
;
```

```
/* ---- white space or comma as separator */
```

```
value_list : real
```

```
{ libc_util_float_list1(&($$), &($1)); }
```

```
| value_list comma. real
```

```
{ libc_util_float_list2(&($$), &($1), &($3)); }
```

```
;
```

```
qvalue_list : ''' value_list '''
```

```
{ $$ = $2; }
```

```
;
```

```
qvalue_lists : qvalue_list
```

```
{ libc_util_float_list_list1(&($$), &($1)); }
```

```
| qvalue_lists comma. qvalue_list
```

```
{ libc_util_float_list_list2(&($$), &($1), &($3)); }
```

```
;
```

```
comma. : ','
```

```
|
```

```
;
```

```
semicolon. : ';'
```

```
|
```

```
;
```

```
sdf_skips : SDF_SKIP
```

```
| sdf_skips SDF_SKIP
```

```
;
```

```
/* ===== */
```

```
library_group : K_LIBRARY '(' name ')'
```

```
{ lex_prev_state();
```

```
tech_lib = new_libc_lib_rec();
```

```
tech_lib->lib_name = $3.string;
```

```
tech_lib->bus_naming_style = copy_string("%s[%d]");
```

```
tech_lib->k_factor = new_libc_k_factor_rec();
```

```

        tech_lib->k_factor->kf_name = NULL;
    }
'{ lib_descriptions '
{ lex_prev_state(); }
semicolon.
;

lib_descriptions      : lib_description
| lib_descriptions lib_description
;

lib_description : def_attr
| lib_simple_attr
| lib_default_attr
| lib_k_attr
| lib_complex_attr
| lib_group_stmt
| ignored_stmt
;

ignored_stmt          : L_DEF_REAL ':' value semicolon.
{ free_text_buffer($1.string); }
| L_DEF_TEXT ':' QSTR_string semicolon.
{ free_text_buffer($1.string);
free_text_buffer($3.string); }
| L_DEF_BOOL ':' boolean semicolon.
{ free_text_buffer($1.string); }
;

lib_simple_attr       : K_ANPPP { libc_time_delay_model(GENERIC_ECL); } ':'
string semicolon.
{ tech_lib->aux_no_pulldown_pin_property = $4.string; }
| K_BNS ':' L_QSTRING semicolon.
{ free_text_buffer(tech_lib->bus_naming_style);
tech_lib->bus_naming_style = $3.string; }
| K_COMMENT ':' L_QSTRING semicolon.
{ tech_lib->comment = $3.string; }
| K_CU ':' cap_unit semicolon.
{ tech_lib->current_unit = $3.unit; }
| K_DATE ':' L_QSTRING semicolon.
{ tech_lib->date = $3.string; }
| K_DM ':' delay_model semicolon.
K_IPSM ':' ipo_type semicolon.
| K_LPU ':' power_unit semicolon.
{ tech_lib->leakage_power_unit = $3.unit; }
| K_MWE ':' L_INT semicolon.
{ tech_lib->max_wired_emitters = $3.int_val; }
| K_MDL ':' boolean semicolon.
{ tech_lib->multiple_drivers_legal = $3.int_val; }
| K_NP ':' value semicolon.
{ tech_lib->nom_process = $3.real_val; }
| K_NT ':' value semicolon.
{ tech_lib->nom_temperature = $3.real_val; }
| K_NV ':' value semicolon.
{ tech_lib->nom_voltage = $3.real_val; }
| K_NTIDF { libc_time_delay_model(GENERIC_ECL); } ':'
mmp_type semicolon.
;
```

libc_yacc.y

```
| K_NPPP { libc_time_delay_model(GENERIC_ECL); } ':' string
semicolon.
| { tech_lib->no_pulldown_pin_property = $4.string; }
| K_PT ':' piece_type semicolon.
| K_PU ':' L_INT power_unit semicolon.
{ tech_lib->power_unit = $3.unit; }
| K_POPSRC ':' string semicolon.
{ tech_lib->preferred_output_pad_slew_rate_control =
$3.string; }
| K_POPV ':' string semicolon.
{ tech_lib->preferred_output_pad_voltage = $3.string; }
| K_PIPV ':' string semicolon.
{ tech_lib->preferred_input_pad_voltage = $3.string; }
| K_PRU ':' res_unit semicolon.
{ tech_lib->pulling_resistance_unit = $3.unit; }
| K_RC ':' value semicolon. /* cmos2 */
{ tech_lib->reference_capacitance = $3.real_val; }
| K_REVISION ':' lib_revision semicolon.
{ tech_lib->revision = $3.string; }
| K_SIMULATION ':' boolean semicolon.
{ tech_lib->simulation = $3.int_val; }
| K_TU ':' time_unit semicolon.
{ tech_lib->time_unit = $3.unit; }
| K_UPP { libc_time_delay_model(GENERIC_ECL); } ':' string
semicolon.
{ tech_lib->unconnected_pin_property = $4.string; }
| K_VU ':' volt_unit semicolon.
{ tech_lib->voltage_unit = $3.unit; }
| K_WLF { libc_time_delay_model(GENERIC_ECL); } ':'
wire_logic_func semicolon.
;

lib_revision : REV_V
;

/* ---- CU */
cap_unit : c_unit
| L_INT c_unit
{ $2.unit->V = (float) $1.int_val;
$$ .unit = $2.unit;
}
| ''' L_INT c_unit '''
{ $3.unit->V = (float) $2.int_val;
$$ .unit = $3.unit;
}
;

c_unit : CU_A { $$ .unit = libc_util_unit(1.0); }
| CU_MA { $$ .unit = libc_util_unit(1.0e-3); }
| CU_UA { $$ .unit = libc_util_unit(1.0e-6); }
;

/* ---- DM */
delay_model : dm_class
| ''' dm_class '''
;
```

libc_yacc.y

```
dm_class          : DM_G_ECL { libc_time_delay_model(GENERIC_ECL); }
| DM_G_CMOS { libc_time_delay_model(GENERIC_CMOS); }
| DM_TBL_LOOKUP { libc_time_delay_model(TABLE_LOOKUP); }
| DM_CMOS2 { libc_time_delay_model(CMOS2); }
| DM_P_COMS { libc_time_delay_model(PIECEWISE_CMOS); }
;

/* ---- IPO */
ipo_type          : ipo_tt
| ''' ipo_tt '''
;

ipo_tt           : IPO_MF { tech_lib->in_place_swap_mode =
MATCH_FOOTPRINT; }
| IPO_NS { tech_lib->in_place_swap_mode = NO_SWAPPING; }
| IPO_IF { tech_lib->in_place_swap_mode =
IGNORE_FOOTPRINT; }
;

/* ---- PU */
power_unit        : p_unit
| L_INT p_unit
{ $2.unit->V = (float) $1.int_val;
 $$ .unit = $2.unit;
}
| ''' L_INT p_unit '''
{ $3.unit->V = (float) $2.int_val;
 $$ .unit = $3.unit;
}
;

p_unit            : PU_W { $$ .unit = libc_util_unit(1.0); }
| PU_MW { $$ .unit = libc_util_unit(1.0e-3); }
| PU_UW { $$ .unit = libc_util_unit(1.0e-6); }
| PU_NW { $$ .unit = libc_util_unit(1.0e-9); }
| PU_PW { $$ .unit = libc_util_unit(1.0e-12); }
;
;

/* ---- BOOL */
boolean           : BOOL_T { $$ .int_val = 1; }
| ''' BOOL_T ''' { $$ .int_val = 1; }
| BOOL_F { $$ .int_val = 0; }
| ''' BOOL_F ''' { $$ .int_val = 0; }
;
;

/* ---- MMP */
mmp_type          : MMP_MAX { tech_lib->nonpaired_twin_inc_delay_func =
MAX_E; }
| MMP_MIN { tech_lib->nonpaired_twin_inc_delay_func =
MIN_E; }
| MMP_PLUS { tech_lib->nonpaired_twin_inc_delay_func =
PLUS_E; }
;
;

/* ---- PT */
piece_type         : PT_LENGTH { tech_lib->piece_type = PIECE_LENGTH; }
;
```

libc_yacc.y

```
| PT_WIRE_CAP      { tech_lib->piece_type = PIECE_WIRE_CAP; }  
}  
| PT_PIN_CAP       { tech_lib->piece_type = PIECE_PIN_CAP; }  
| PT_TOTAL_CAP     { tech_lib->piece_type = PIECE_TOTAL_CAP; }  
}  
;  
  
/* ---- RU */  
res_unit : r_unit  
| L_INT r_unit  
{ $2.unit->V = (float) $1.int_val;  
 $$ .unit = $2.unit;  
}  
| ''' L_INT r_unit '''  
{ $3.unit->V = (float) $2.int_val;  
 $$ .unit = $3.unit;  
}  
;  
  
r_unit : RU_OHM    { $$ .unit = libc_util_unit(1.0); }  
| RU_KOHM   { $$ .unit = libc_util_unit(1.0e3); }  
;  
  
/* ---- TU */  
time_unit : t_unit  
| L_INT t_unit  
{ $2.unit->V = (float) $1.int_val;  
 $$ .unit = $2.unit;  
}  
| ''' L_INT t_unit '''  
{ $3.unit->V = (float) $2.int_val;  
 $$ .unit = $3.unit;  
}  
;  
  
t_unit : TU_NS      { $$ .unit = libc_util_unit(1.0e-9); }  
| TU_PS      { $$ .unit = libc_util_unit(1.0e-12); }  
;  
  
/* ---- VU */  
volt_unit : v_unit  
| L_INT v_unit  
{ $2.unit->V = (float) $1.int_val;  
 $$ .unit = $2.unit;  
}  
| ''' L_INT v_unit '''  
{ $3.unit->V = (float) $2.int_val;  
 $$ .unit = $3.unit;  
}  
;  
  
v_unit : VU_V       { $$ .unit = libc_util_unit(1.0); }  
| VU_MV      { $$ .unit = libc_util_unit(1.0e-3); }  
;  
  
/* ---- WLF */  
wire_logic_func : WLF_WAND { tech_lib->wired_logic_function = WIRED_AND; }
```

```

| WLF_WOR    { tech_lib->wired_logic_function = WIRED_OR; }
;

/* ----- */

lib_default_attr : K_DCLP ':' value semicolon. { tech_lib-
>default_cell_leakage_power = $3.real_val; }
| K_DCP ':' value semicolon. { tech_lib-
>default_cell_power = $3.real_val; }
| K_DCC ':' name_list ';' { tech_lib-
>default_connection_class = $3.name_list.head; }
| K_DF0 { libc_time_delay_model(CMOS2); } ':' value
semicolon.
{ tech_lib->default_edge_rate_breakpoint_f0 = $4.real_val;
}
| K_DF1 { libc_time_delay_model(CMOS2); } ':' value
semicolon.
{ tech_lib->default_edge_rate_breakpoint_f1 = $4.real_val;
}
| K_DR0 { libc_time_delay_model(CMOS2); } ':' value
semicolon.
{ tech_lib->default_edge_rate_breakpoint_r0 = $4.real_val;
}
| K_DR1 { libc_time_delay_model(CMOS2); } ':' value
semicolon.
{ tech_lib->default_edge_rate_breakpoint_r1 = $4.real_val;
}
| K_DEC ':' L_INT semicolon. { tech_lib-
>default_emitter_count = $3.int_val; }
| K_DFDI { libc_time_delay_model(PIECEWISE_CMOS); } ':' value
semicolon.
{ tech_lib->default_fall_delay_intercept = $4.real_val; }
| K_DFNT { libc_time_delay_model(PIECEWISE_CMOS); } ':' value
semicolon.
{ tech_lib->default_fall_nonpaired_twin = $4.real_val; }
| K_DFPR { libc_time_delay_model(PIECEWISE_CMOS); } ':' value
semicolon.
{ tech_lib->default_fall_pin_resistance = $4.real_val; }
| K_DFWR ':' value semicolon. { tech_lib-
>default_fall_wire_resistance = $3.real_val; }
| K_DFWE ':' value semicolon. { tech_lib-
>default_fall_wor_emitter = $3.real_val; }
| K_DFWI ':' value semicolon. { tech_lib-
>default_fall_wor_intercept = $3.real_val; }
| K_DFL ':' value semicolon. { tech_lib-
>default_fanout_load = $3.real_val; }
| K_DIOPC ':' value semicolon. { tech_lib-
>default_inout_pin_cap = $3.real_val; }
| K_DIOFR ':' value semicolon. { tech_lib-
>default_inout_pin_fall_res = $3.real_val; }
| K_DIORR ':' value semicolon. { tech_lib-
>default_inout_pin_rise_res = $3.real_val; }
| K_DIPC ':' value semicolon. { tech_lib-
>default_input_pin_cap = $3.real_val; }
| K_DIF ':' value semicolon. { tech_lib-
>default_intrinsic_fall = $3.real_val; }

```

```

        | K_DIR ':' value semicolon. { tech_lib-
>default_intrinsic_rise = $3.real_val; }
        | K_DMC ':' value semicolon. { tech_lib-
>default_max_capacitance = $3.real_val; }
        | K_DMFO ':' value semicolon. { tech_lib-
>default_max_fanout = $3.real_val; }
        | K_DMT ':' value semicolon. { tech_lib-
>default_max_transition = $3.real_val; }
        | K_DMU ':' value semicolon. { tech_lib-
>default_max_utilization = $3.real_val; }
        | K_DMP ':' value semicolon. { tech_lib-
>default_min_porosity = $3.real_val; }
        | K_DOC ':' name semicolon. { tech_lib-
>default_operating_conditions = $3.string; }
        | K_DOPC ':' value semicolon. { tech_lib-
>default_output_pin_cap = $3.real_val; }
        | K_DOFR ':' value semicolon. { tech_lib-
>default_output_pin_fall_res = $3.real_val; }
        | K_DORR ':' value semicolon. { tech_lib-
>default_output_pin_rise_res = $3.real_val; }
        | K_DPL ':' L_INT semicolon. { tech_lib->default_pin_limit
= $3.real_val; }
        | K_DPP ':' L_INT semicolon. { tech_lib->default_pin_power
= $3.real_val; }
        | K_DRFC ':' value semicolon. { tech_lib-
>default_rc_fall_coefficient = $3.real_val; }
        | K_DRRC ':' value semicolon. { tech_lib-
>default_rc_rise_coefficient = $3.real_val; }
        | K_DRDI { libc_time_delay_model(PIECEWISE_CMOS); } ':'
value semicolon.
        { tech_lib->default_rise_delay_intercept = $4.real_val; }
        | K_DRNT { libc_time_delay_model(PIECEWISE_CMOS); } ':'
value semicolon.
        { tech_lib->default_rise_nonpaired_twin = $4.real_val; }
        | K_DRPR { libc_time_delay_model(PIECEWISE_CMOS); } ':'
value semicolon.
        { tech_lib->default_rise_pin_resistance = $4.real_val; }
        | K_DRWR ':' value semicolon. { tech_lib-
>default_rise_wire_resistance = $3.real_val; }
        | K_DRWE ':' value semicolon. { tech_lib-
>default_rise_wor_emitter = $3.real_val; }
        | K_DRWI ':' value semicolon. { tech_lib-
>default_rise_wor_intercept = $3.real_val; }
        | K_DSC ':' value semicolon. { tech_lib-
>default_setup_coefficient = $3.real_val; }
        | K_DSF ':' value semicolon. { tech_lib-
>default_slope_fall = $3.real_val; }
        | K_DSR ':' value semicolon. { tech_lib-
>default_slope_rise = $3.real_val; }
        | K_DWL ':' name
        { lex_prev_state(); tech_lib->default_wire_load =
$3.string; } semicolon.
        | K_DWLA ':' value semicolon. { tech_lib-
>default_wire_load_area = $3.real_val; }
        | K_DWL_C ':' value semicolon. { tech_lib-
>default_wire_load_capacitance = $3.real_val; }
        | K_DWL_M ':' wire_load_mode semicolon.

```

libc_yacc.y

```
| K_DWLR '::' value semicolon. { tech_lib->default_wire_load_resistance = $3.real_val; }
| K_DWLS '::' name
{ lex_prev_state(); tech_lib->default_wire_load_selection = $3.string; } semicolon.
;

/* ---- WLM */
wire_load_mode : WLM_T          { tech_lib->default_wire_load_mode =
TOP_WL; }
| WLM_S           { tech_lib->default_wire_load_mode =
SEGMENTED_WL; }
| WLM_E           { tech_lib->default_wire_load_mode =
ENCLOSED_WL; }
;

/*
----- */

lib_k_attr : P_CR { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
{ kfc = KF(); kfc->k_cell_rise[scaling_attr] = $4.real_val; }
| P_CF { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
{ kfc = KF(); kfc->k_cell_fall[scaling_attr] = $4.real_val; }
| P_CLP '::' value semicolon. { kfc = KF(); kfc->k_cell_leakage_power[scaling_attr] = $3.real_val; }
| P_CP '::' value semicolon. { kfc = KF(); kfc->k_cell_power[scaling_attr] = $3.real_val; }
| P_DC '::' value semicolon. { kfc = KF(); kfc->k_drive_current[scaling_attr] = $3.real_val; }
| P_DF '::' value semicolon. { kfc = KF(); kfc->k_drive_fall[scaling_attr] = $3.real_val; }
| P_DR '::' value semicolon. { kfc = KF(); kfc->k_drive_rise[scaling_attr] = $3.real_val; }
| P_FDI { libc_time_delay_model(PIECEWISE_CMOS); } '::' value
semicolon.
{ kfc = KF(); kfc->k_fall_delay_intercept[scaling_attr] =
$4.real_val; }
| P_FPR { libc_time_delay_model(PIECEWISE_CMOS); } '::' value
semicolon.
{ kfc = KF(); kfc->k_fall_pin_resistance[scaling_attr] =
$4.real_val; }
| P_FP { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
{ kfc = KF(); kfc->k_fall_propagation[scaling_attr] =
$4.real_val; }
| P_FT { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
{ kfc = KF(); kfc->k_fall_transition[scaling_attr] =
$4.real_val; }
| P_FWR '::' value semicolon. { kfc = KF(); kfc->k_fall_wire_resistance[scaling_attr] = $3.real_val; }
| P_FWE '::' value semicolon. { kfc = KF(); kfc->k_fall_wor_emitter[scaling_attr] = $3.real_val; }
| P_FWI '::' value semicolon. { kfc = KF(); kfc->k_fall_wor_intercept[scaling_attr] = $3.real_val; }
```

```

        | P_HF '::' value semicolon. { kfc = KF(); kfc-
>k_hold_fall[scaling_attr] = $3.real_val; }
        | P_HR '::' value semicolon. { kfc = KF(); kfc-
>k_hold_rise[scaling_attr] = $3.real_val; }
        | P_IP '::' value semicolon. { kfc = KF(); kfc-
>k_internal_power[scaling_attr] = $3.real_val; }
        | P_IF '::' value semicolon. { kfc = KF(); kfc-
>k_intrinsic_fall[scaling_attr] = $3.real_val; }
        | P_IR '::' value semicolon. { kfc = KF(); kfc-
>k_intrinsic_rise[scaling_attr] = $3.real_val; }
        | P_MP '::' value semicolon. { kfc = KF(); kfc-
>k_min_period[scaling_attr] = $3.real_val; }
        | P_MPWH '::' value semicolon. { kfc = KF(); kfc-
>k_min_pulse_width_high[scaling_attr] = $3.real_val; }
        | P_MPWL '::' value semicolon. { kfc = KF(); kfc-
>k_min_pulse_width_low[scaling_attr] = $3.real_val; }
        | P_NF '::' value semicolon. { kfc = KF(); kfc-
>k_nochange_fall[scaling_attr] = $3.real_val; }
        | P_NR '::' value semicolon. { kfc = KF(); kfc-
>k_nochange_rise[scaling_attr] = $3.real_val; }
        | P_PC '::' value semicolon. { kfc = KF(); kfc-
>k_pin_cap[scaling_attr] = $3.real_val; }
        | P_PP '::' value semicolon. { kfc = KF(); kfc-
>k_pin_power[scaling_attr] = $3.real_val; }
        | P_RF '::' value semicolon. { kfc = KF(); kfc-
>k_recovery_fall[scaling_attr] = $3.real_val; }
        | P_RR '::' value semicolon. { kfc = KF(); kfc-
>k_recovery_rise[scaling_attr] = $3.real_val; }
        | P_REF '::' value semicolon. { kfc = KF(); kfc-
>k_removal_fall[scaling_attr] = $3.real_val; }
        | P_RER '::' value semicolon. { kfc = KF(); kfc-
>k_removal_rise[scaling_attr] = $3.real_val; }
        | P_RDI { libc_time_delay_model(PIECEWISE_CMOS); } '::' value
semicolon.
        { kfc = KF(); kfc->k_rise_delay_intercept[scaling_attr] =
$4.real_val; }
        | P_RPR { libc_time_delay_model(PIECEWISE_CMOS); } '::' value
semicolon.
        { kfc = KF(); kfc->k_rise_pin_resistance[scaling_attr] =
$4.real_val; }
        | P_RP { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
        { kfc = KF(); kfc->k_rise_propagation[scaling_attr] =
$4.real_val; }
        | P_RT { libc_time_delay_model(TABLE_LOOKUP); } '::' value
semicolon.
        { kfc = KF(); kfc->k_rise_transition[scaling_attr] =
$4.real_val; }
        | P_RWR '::' value semicolon. { kfc = KF(); kfc-
>k_rise_wire_resistance[scaling_attr] = $3.real_val; }
        | P_RWE '::' value semicolon. { kfc = KF(); kfc-
>k_rise_wor_emitter[scaling_attr] = $3.real_val; }
        | P_RWI '::' value semicolon. { kfc = KF(); kfc-
>k_rise_wor_intercept[scaling_attr] = $3.real_val; }
        | P_SF '::' value semicolon. { kfc = KF(); kfc-
>k_setup_fall[scaling_attr] = $3.real_val; }

```

```

    | P_SR '::' value semicolon. { kfc = KF(); kfc-
>k_setup_rise[scaling_attr] = $3.real_val; }
    | P_SKF '::' value semicolon. { kfc = KF(); kfc-
>k_skew_fall[scaling_attr] = $3.real_val; }
    | P_SKR '::' value semicolon. { kfc = KF(); kfc-
>k_skew_rise[scaling_attr] = $3.real_val; }
    | P_SLF '::' value semicolon. { kfc = KF(); kfc-
>k_slope_fall[scaling_attr] = $3.real_val; }
    | P_SLR '::' value semicolon. { kfc = KF(); kfc-
>k_slope_rise[scaling_attr] = $3.real_val; }
    | P_WC '::' value semicolon. { kfc = KF(); kfc-
>k_wire_cap[scaling_attr] = $3.real_val; }
    | P_WR '::' value semicolon. { kfc = KF(); kfc-
>k_wire_res[scaling_attr] = $3.real_val; }
;

/* ----- */

lib_complex_attr : K_CLU '(' value ',' load_unit ')' semicolon.
{ $5.unit->V = $3.real_val;
  tech_lib->capacitive_load_unit = $5.unit;
}
| K_DEFINE '(' string ',' string ',' string ')' semicolon.
{ libc_def_insert($3.string,$5.string,$7.string); }
| K_DCA '(' string ',' pad_type ')' semicolon.
{ libc_def_cell_area($3.string,$5.pad_type); }
| K_LF '(' qvalue_list ')' semicolon.
{ free_libc_float_list_rec($3.float_list.head); }
| K_LF '(' name_list ')' semicolon.
{ free_libc_name_list_rec($3.name_list.head); }
| K_PD '(' qvalue_list ')' semicolon.
{ tech_lib->piece_define =
  libc_util_float_list2buffer($3.float_list.head); }
| K_RL '(' name_list ')' semicolon.
{ tech_lib->routing_layers = $3.name_list.head; }
| K_TECH '(' name ')'
{ lex_prev_state(); } semicolon.
{ tech_lib->technology = $3.string; }
;

/* ---- LU */
load_unit : LU_MF      { $$ .unit = libc_util_unit(1.0e-3); }
| LU_UF      { $$ .unit = libc_util_unit(1.0e-6); }
| LU_NF      { $$ .unit = libc_util_unit(1.0e-9); }
| LU_PF      { $$ .unit = libc_util_unit(1.0e-12); }
| LU_FF      { $$ .unit = libc_util_unit(1.0e-15); }
;

/* ---- RT */
pad_type : RT_PS      { $$ .pad_type = PAD_SLOTS; }
| RT_PDS     { $$ .pad_type = PAD_DRIVER_SITES; }
| RT_PIDS    { $$ .pad_type = PAD_INPUT_DRIVER_SITES; }
| RT_PODS    { $$ .pad_type = PAD_OUTPUT_DRIVER_SITES; }
;

/* ----- */

lib_group_stmt : cell_group

```

```

    | input_volt_group
    | lu_table_group
    | op_cond_group
    | output_volt_group
    | para_cell_group
    | power_lut_group
    | power_supply_group
    | trans_group
    | scaled_cell_group
    | scaled_factor_group
    | timing_range_group
    | type_group
    | wire_load_group
    | wire_load_sel_group
    | wire_load_table_group
;

/* ===== */
/* ---- ignore */
input_volt_group : K_IV '(' name ')'
    { free_text_buffer($3.string); lex_prev_state(); }
    '{' iv_stmts '}'
    { lex_prev_state(); }
semicolon.

iv_stmts : iv_stmt
| iv_stmts iv_stmt
;

iv_stmt : IV_L ':' n_expression semicolon.
| IV_H ':' n_expression semicolon.
| IV_MIN ':' n_expression semicolon.
| IV_MAX ':' n_expression semicolon.
;

/* ---- ignore */
output_volt_group : K_OV '(' name ')'
    { free_text_buffer($3.string); lex_prev_state(); }
    '{' ov_stmts '}'
    { lex_prev_state(); }
semicolon.

ov_stmts : ov_stmt
| ov_stmts ov_stmt
;

ov_stmt : OV_L ':' n_expression semicolon.
| OV_H ':' n_expression semicolon.
| OV_MIN ':' n_expression semicolon.
| OV_MAX ':' n_expression semicolon.
;

/* ----- */

```

```

lu_table_group      : K_LTT '(' name ')'
{ lex_prev_state();
  lu_table_temp = new_libc_lu_table_template_rec();
  lu_table_temp->tt_name = $3.string;
}
'{ lut_stmts }'
{ lex_prev_state();
  lu_table_temp->next = tech_lib->lut_template;
  tech_lib->lut_template = lu_table_temp;
  lu_table_temp = NULL;
}
semicolon.
;

lut_stmts          : lut_stmt
| lut_stmts lut_stmt
;

lut_stmt           : TBL_VAR1 ':' var_type semicolon.
{ lu_table_temp->variable_1 = $3.var_type; }
| TBL_VAR2 ':' var_type semicolon.
{ lu_table_temp->variable_2 = $3.var_type; }
| TBL_VAR3 ':' var_type semicolon.
{ lu_table_temp->variable_3 = $3.var_type; }
| TBL_IDX1 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lu_table_temp->index_1);
  lu_table_temp->index_1 =
  libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX2 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lu_table_temp->index_2);
  lu_table_temp->index_2 =
  libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX3 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lu_table_temp->index_3);
  lu_table_temp->index_3 =
  libc_util_float_list2buffer($3.float_list.head); }
;
var_type           : LTT_INT   { $$.var_type = INPUT_NET_TRANSITION; }
| LTT_TONC   { $$.var_type = TOTAL_OUTPUT_NET_CAPACITANCE; }
| LTT_ONL    { $$.var_type = OUTPUT_NET_LENGTH; }
| LTT_ONWC   { $$.var_type = OUTPUT_NET_WIRE_CAP; }
| LTT_ONPC   { $$.var_type = OUTPUT_NET_PIN_CAP; }
| LTT_ROTONC { $$.var_type =
RELATED_OUT_TOTAL_OUTPUT_NET_CAP; }
| LTT_ROONL  { $$.var_type = RELATED_OUT_OUTPUT_NET_LENGTH;
}
| LTT_ROONWC { $$.var_type =
RELATED_OUT_OUTPUT_NET_WIRE_CAP; }
| LTT_ROONPC { $$.var_type =
RELATED_OUT_OUTPUT_NET_PIN_CAP; }
| LTT_CPT    { $$.var_type = CONSTRAINED_PIN_TRANSITION; }
| LTT_RPT    { $$.var_type = RELATED_PIN_TRANSITION; }
| LTT_OPT    { $$.var_type = OUTPUT_PIN_TRANSITION; }
| LTT_CD     { $$.var_type = CONNECT_DELAY; }
| string
{ $$.var_type = INPUT_NET_TRANSITION;
}

```

```

        libcerror("un-recognized variable type!");
    }
;

/* ----- */

op_cond_group      : K_OC '(' name ')'
{
    lex_prev_state();
    lib_op_cond = new_libc_operating_condition_rec();
    lib_op_cond->oc_name = $3.string;
}
'{' oc_stmts '}'
{
    lex_prev_state();
    lib_op_cond->next = tech_lib->operating_cond;
    tech_lib->operating_cond = lib_op_cond;
    lib_op_cond = NULL;
}
semicolon.
;

oc_stmts          : oc_stmt
| oc_stmts oc_stmt
;

oc_stmt           : OC_PROCESS '::' value semicolon.
{
    lib_op_cond->process = $3.real_val; }
| OC_TEMP '::' value semicolon.
{
    lib_op_cond->temperature = $3.real_val; }
| OC_TREE '::' tree_type semicolon.
| OC_VOLT '::' value semicolon.
{
    lib_op_cond->voltage = $3.real_val; }
| power_rail_attr
{
    $1.prp->next = lib_op_cond->power_rail;
    lib_op_cond->power_rail = $1.prp;
}
;

power_rail_attr : OC_PR '(' string ',' value ')'
{
    $$ .prp = new_libc_oc_power_rail_rec();
    $$ .prp->power_supply = $3.string;
    $$ .prp->volt       = $5.real_val;
}
;

/* ---- TT */
tree_type         : tree_tt
| ''' tree_tt '''
;

tree_tt          : TT_BEST   { lib_op_cond->tree_type = BEST_CASE_TREE; }
| TT_BAL     { lib_op_cond->tree_type = BALANCED_TREE; }
| TT_WORST   { lib_op_cond->tree_type = WORST_CASE_TREE; }
;
;

/* ----- */

```

libc_yacc.y

```
power_supply_group      : K_PS '(' name ')'
{ lex_prev_state();
  lib_ps = new_libc_power_supply_rec();
  lib_ps->ps_name = $3.string;
}
'{ ps_stmts}'
{ lex_prev_state();
  lib_ps->next = tech_lib->power_supply;
  tech_lib->power_supply = lib_ps;
  lib_ps = NULL;
}
semicolon.
;

ps_stmts               : ps_stmt
| ps_stmts ps_stmt
;

ps_stmt                : PS_DPR ':' string ;
{ lib_ps->default_ps = $3.string; }
| power_rail_attr
{ $1.prp->next = lib_ps->power_rail;
  lib_ps->power_rail = $1.prp;
}
;
/* ----- */

/* ---- ignored */
para_cell_group         : K_PCELL '(' name ')'
{ lex_prev_state(); }
'{ pcell_stmts}'
{ lex_prev_state(); }
semicolon.
;

pcell_stmts             : pcell_stmt
| pcell_stmts pcell_stmt
;

pcell_stmt              : cell_simple_attr
| cell_complex_attr
| cell_enum_group
;
;

cell_enum_group          : PC_CELL_ENUM '(' name ')'
{ lex_prev_state(); }
'{ ce_stmts}'
{ lex_prev_state(); }
semicolon.
;

ce_stmts                : ce_stmt
| ce_stmts ce_stmt
;
;

ce_stmt                 : cell_simple_attr
```

```

    | cell_complex_attr
    | CE_PROPERTY ':' string semicolon.
    | CE_DE ':' boolean semicolon.
    bus_group
    test_cell_group
    para_pin_group
;

para_pin_group : CE_PP '(' name_list ')' '{' pp_stmts '}'
{ lex_prev_state(); }
semicolon.
;

pp_stmts      : pp_stmt
| pp_stmts pp_stmt
;

pp_stmt       : pin_simple_attr
| PP_PROPERTIES ':' '{' name_list '}' semicolon.
| timing_group
;

/*
----- */

power_lut_group : K_PLT '(' name ')'
{ lex_prev_state();
plut_temp = new_libc_lu_table_template_rec();
plut_temp->tt_name = $3.string;
}
'{' plut_stmts '}'
{ lex_prev_state();
plut_temp->next = tech_lib->plut_template;
tech_lib->plut_template = plut_temp;
plut_temp = NULL;
}
semicolon.
;

plut_stmts    : plut_stmt
| plut_stmts plut_stmt
;

plut_stmt     : TBL_VAR1 ':' var_type semicolon.
{ plut_temp->variable_1 = $3.var_type; }
| TBL_VAR2 ':' var_type semicolon.
{ plut_temp->variable_2 = $3.var_type; }
| TBL_VAR3 ':' var_type semicolon.
{ plut_temp->variable_3 = $3.var_type; }
| TBL_IDX1 '(' qvalue_list ')' semicolon.
{ free_float_buffer(plut_temp->index_1);
plut_temp->index_1 =
libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX2 '(' qvalue_list ')' semicolon.
{ free_float_buffer(plut_temp->index_2);
plut_temp->index_2 =
libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX3 '(' qvalue_list ')' semicolon.
;
```

libc_yacc.y

```
        { free_float_buffer(plut_temp->index_3);
          plut_temp->index_3 =
    libc_util_float_list2buffer($3.float_list.head); }
        ;

/* ----- */

/* ---- rise_transition_degradation, fall_transition_degradation group */
trans_group : trans_type '(' name
        { lex_prev_state();
          lib_timing_tbl = new_libc_table_val_rec();
          if (!libc_time_find_template($3.string,tech_lib-
>lut_template,lib_timing_tbl)) {
              libcerror("template not found!");
              exit(1);
          }
        }
        ')' '{' VALUES '(' qvalue_lists ')' semicolon.  '}'
        { lex_prev_state(); }

libc_time_values_handle(lib_timing_tbl,$9.float_list_list.head);
        if ($1.int_val) {
            free_libc_table_val_rec(tech_lib->rise_tr_table);
            tech_lib->rise_tr_table = lib_timing_tbl;
        }
        else {
            free_libc_table_val_rec(tech_lib->fall_tr_table);
            tech_lib->fall_tr_table = lib_timing_tbl;
        }
        lib_timing_tbl = NULL;
    }
    semicolon.
;

trans_type : K_RTD      { $$.int_val = 1; }
| K_FTD      { $$.int_val = 0; }
;

/* ----- */

/* ---- ignored */
scaled_cell_group : K_SC '(' name ',' name ')'
        { lex_prev_state();
          libc_cell_init($3.string);
        }
        '{' cell_stmts '}'
        { lex_prev_state();
          free_text_buffer($5.string);
          free_libc_cell_rec(lib_cell);
          lib_cell = NULL;
        }
;

/* ----- */

/* ---- different k factors for different cell */
scaled_factor_group : K_SF '(' name ')'
```

```

    { lex_prev_state();
      sc_kfc = new_libc_k_factor_rec();
      sc_kfc->kf_name = $3.string;
    }
    '{' sf_stmts '}')
    { lex_prev_state();
      sc_kfc->next = tech_lib->sc_k_factor;
      tech_lib->sc_k_factor = sc_kfc;
      sc_kfc = NULL;
    }
    semicolon.
;

sf_stmts : sf_stmt
| sf_stmts sf_stmt
;

sf_stmt : lib_k_attr /* memory leak here ? */
;

/* ----- */

timing_range_group : K_TR '(' name ')'
{ lex_prev_state();
  lib_timing_range = new_libc_timing_range_rec();
  lib_timing_range->tr_name = $3.string;
}
'{' tr_stmts '}')
{ lex_prev_state();
  lib_timing_range->next = tech_lib->timing_range;
  tech_lib->timing_range = lib_timing_range;
  lib_timing_range = NULL;
}
semicolon.
;

tr_stmts : tr_stmt
| tr_stmts tr_stmt
;

tr_stmt : TR_FF '::' value semicolon.
| TR_SF '::' value semicolon.
{ lib_timing_range->faster_factor = $3.real_val; }
{ lib_timing_range->slower_factor = $3.real_val; }
;

/* ----- */

type_group : K_TYPE '(' name ')'
{ lex_prev_state();
  lib_type_name = new_libc_type_rec();
  lib_type_name->type_name = $3.string;
}
'{' type_stmts '}')
{ lex_prev_state();
  lib_type_name->next = tech_lib->type;
  tech_lib->type = lib_type_name;
}
;
```

```

        lib_type_name = NULL;
    }
semicolon.
;

type_stmts      : type_stmt
| type_stmts type_stmt
;

type_stmt       : TYPE_BASE ':' type_array semicolon.
| TYPE_FROM ':' L_INT semicolon.
{ lib_type_name->bit_from = $3.int_val; }
| TYPE_TO ':' L_INT semicolon.
{ lib_type_name->bit_to = $3.int_val; }
| TYPE_WIDTH ':' L_INT semicolon.
{ lib_type_name->bit_width = $3.int_val; }
| TYPE_DT ':' type_bit semicolon.
| TYPE_DOWNT0 ':' boolean semicolon.
{ lib_type_name->downto = $3.int_val; }
;

type_array      : TYPE_ARRAY
| """ TYPE_ARRAY """
;

type_bit        : TYPE_BIT
| """ TYPE_BIT """
;

/* ----- */

wire_load_group : K_WL '(' name ')'
{ lex_prev_state();
  lib_wire_load = new_libc_wire_load_rec();
  lib_wire_load->wl_name = $3.string;
  lib_wire_load->area      = tech_lib-
>default_wire_load_area;
  lib_wire_load->capacitance = tech_lib-
>default_wire_load_capacitance;
  lib_wire_load->resistance  = tech_lib-
>default_wire_load_resistance;
}
'{' wl_stmt1s wl_stmt2s '}'
{ lex_prev_state();
  lib_wire_load->next = tech_lib->wire_load;
  tech_lib->wire_load = lib_wire_load;
  lib_wire_load = NULL;
}
semicolon.
;

wl_stmt1s.      : wl_stmt1s
|
;

wl_stmt1s       : wl_stmt1
| wl_stmt1s wl_stmt1
;
```

```

;

wl_stmt1      : AREA ':' value semicolon.
| lib_wire_load->area = $3.real_val; }
| PIN_CAP ':' value semicolon.
| lib_wire_load->capacitance = $3.real_val; }
| WL_RES ':' value semicolon.
| lib_wire_load->resistance = $3.real_val; }
| WL_SLOPE ':' value semicolon.
| lib_wire_load->slope = $3.real_val; }
;

wl_stmt2s     : wl_stmt2
| wl_stmt2s wl_stmt2
;

wl_stmt2      : WL_FL '(' value ',' value ')' semicolon.          /* <-
--- old form */
{
    libc_util_wire_load_fanout((int)$3.real_val,$5.real_val,0.0,0.0,0);
    | WL_FL ':' '(' L_INT ',' value ',' value ',' value ',' 
L_INT ')' semicolon.
    {
        libc_util_wire_load_fanout($4.int_val,$6.real_val,$8.real_val,$10.real_val,$1
2.int_val); }
    ;

/* ----- */

wire_load_sel_group : K_WLS '(' name.
{
    lex_prev_state();
')
{
    lib_wire_load_sel = new_libc_wire_load_selection_rec();
    lib_wire_load_sel->wls_name = $3.string;
}
'{ wls_stmts '}'
{
    lex_prev_state();
    lib_wire_load_sel->next      = tech_lib-
>wire_load_selection;
    tech_lib->wire_load_selection = lib_wire_load_sel;
    lib_wire_load_sel            = NULL;
}
semicolon.
;

wls_stmts     : wls_stmt
| wls_stmts wls_stmt
;

wls_stmt      : WLS_WLFA '(' value ',' value ','
{
    lex_next_NAME_state();
}
name
{
    lex_prev_state();
')
')' semicolon.
{
    libc_util_wl_select($3.real_val,$5.real_val,$8.string);
}
;

```

libc_yacc.y

```
/* ----- */

/* ---- ignore */
wire_load_table_group : K_WLT '(' name ')'
{ lex_prev_state();
  lib_wire_load = new_libc_wire_load_rec();
  lib_wire_load->wl_name = $3.string;
  lib_wire_load->area      = tech_lib-
>default_wire_load_area;
  lib_wire_load->capacitance = tech_lib-
>default_wire_load_capacitance;
  lib_wire_load->resistance  = tech_lib-
>default_wire_load_resistance;
}
'{ wlt_stmts }'
{ lex_prev_state();
  lib_wire_load->next = tech_lib->wire_load;
  tech_lib->wire_load = lib_wire_load;
  lib_wire_load = NULL;
}
semicolon.
;

wlt_stmts : wlt_stmt
| wlt_stmts wlt_stmt
;

wlt_stmt : WL_FL colon. '(' L_INT ',' value ')' semicolon.
{ libc_util_wire_load_table($4.int_val,1,$6.real_val); }
| WLT_CAP colon. '(' L_INT ',' value ')' semicolon.
{ libc_util_wire_load_table($4.int_val,2,$6.real_val); }
| WLT_RES colon. '(' L_INT ',' value ')' semicolon.
{ libc_util_wire_load_table($4.int_val,3,$6.real_val); }
| WLT_AREA colon. '(' L_INT ',' value ')' semicolon.
{ libc_util_wire_load_table($4.int_val,4,$6.real_val); }
;
colon.   : ':'
|
;

/* ===== */

cell_group : K_CELL '(' name ')'
{ lex_prev_state();
  libc_cell_init($3.string);
}
'{ cell_stmts }'
{ lex_prev_state();
  lib_cell->next = tech_lib->cells;
  tech_lib->cells = lib_cell;
  lib_cell = NULL;
}
semicolon.
;

cell_stmts : cell_stmt
```

```

| cell_stmts cell_stmt
;

cell_stmt : cell_simple_attr
| cell_complex_attr
| cell_group_stmt
| ignored_stmt
;

/* ----- */

cell_simple_attr : AREA '::' value semicolon.
{ lib_cell->area = $3.real_val; }
| C_APC '::' boolean semicolon.
{ lib_cell->auxiliary_pad_cell = $3.int_val; }
| C_CF '::' string { lex_prev_state(); } semicolon.
{ lib_cell->cell_footprint = $3.string; }
| C_POWER '::' value semicolon.
{ lib_cell->cell_power = $3.real_val; }
| C_LP '::' value semicolon.
{ lib_cell->cell_leakage_power = $3.real_val; }
| C_CC '::' bool_expression semicolon.
{ lib_cell->contention_condition = $3.bool_opr; }
| C_DF '::' stuck_at_type semicolon.
{ lib_cell->dont_false = $3.sa_type; }
| C_DT '::' boolean semicolon.
{ lib_cell->dont_touch = $3.int_val; }
| C_GP '::' string semicolon.
{ lib_cell->geometry_print = $3.string; }
| C_DU '::' boolean semicolon.
{ lib_cell->dont_use = $3.int_val; }
| C_HNC '::' boolean semicolon.
{ lib_cell->handle_negative_constraint = $3.int_val; }
| C_IT '::' boolean semicolon. /* old */
{ lib_cell->interface_timing = $3.int_val; }
| C_MO '::' boolean semicolon.
{ lib_cell->map_only = $3.int_val; }
| C_PC '::' boolean semicolon.
{ lib_cell->pad_cell = $3.int_val; }
| C_PT '::' string semicolon. /* string must be clock
*/
{
    lib_cell->pad_type = !strcmp("clock", $3.string);
    free_text_buffer($3.string);
}
| C_PL { libc_time_delay_model(GENERIC_ECL); } ':' L_INT
semicolon.

{ lib_cell->pin_limit = $4.int_val; }
| C_P '::' boolean semicolon.
{ lib_cell->preferred = $3.int_val; }
| C_SF '::' name semicolon.
{ lib_cell->scaling_factors =
libc_cell_find_sc_group($3.string); }
| C_SG '::' QSTR_string semicolon.
{ lib_cell->scan_group = $3.string; }
| C_SBD '::' string semicolon.
{ lib_cell->single_bit_degenerate = $3.string; }
| C_VN '::' string semicolon.

```

```

    { lib_cell->vhdl_name = $3.string; }
;

stuck_at_type      : SA_SA0      { $$sa_type = SA0; }
| SA_SA1      { $$sa_type = SA1; }
| SA_SA01     { $$sa_type = SA01; }
;

/* ----- */

cell_complex_attr : C_PE '(' qname_list ')' semicolon.
    { libc_cell_name_list_list(&(lib_cell-
>pin_equal),$3.name_list.head,NULL); }
    | C_PO '(' qname_list ',' qname_list ')' semicolon.
    { libc_cell_name_list_list(&(lib_cell-
>pin_opposite),$3.name_list.head,$5.name_list.head); }
    | C_RC '(' string ',' string ')'
        { free_text_buffer($3.string);
free_text_buffer($5.string); }
;
;

/* ----- */

cell_group_stmt      : bundle_group
| bus_group
| ff_group
| ff_bank_group
| internal_power_group_c
| leakage_power_group
| latch_group
| latch_bank_group
| lut_group
| memory_group
| pin_group
| routing_track_group
| state_group
| statetable_group
| test_cell_group
;
;

/* ===== */

bundle_group      : C_BUNDLE '(' name ')'
{ libc_name_list_rec *nl;
nl = new_libc_name_list_rec();
nl->name = $3.string;
libc_cell_pin_init(nl);
}
'{ members_stmt bundle_stmts '}'
{ lex_prev_state();
libc_cell_bundle_post(lib_pin);
lib_pin->next = lib_cell->pins;
lib_cell->pins = lib_pin;
lib_pin = NULL;
}
semicolon.
;
;
```

libc_yacc.y

```
members_stmt      : MEMBERS '(' name_list ')' semicolon.
{ lib_pin->members = $3.name_list.head; }
;

bundle_stmts     : bundle_stmt
| bundle_stmts bundle_stmt
;

bundle_stmt      : pin_simple_attr
| pin_group_stmt
| pin_group
{ lib_pin = lib_bundle;
  lib_bundle = NULL;
}
;

/* ----- */

bus_group         : C_BUS '(' name ')'
{ libc_name_list_rec *nl;
  nl = new_libc_name_list_rec();
  nl->name = $3.string;
  libc_cell_pin_init(nl);
  lib_pin->is_bus = 1;
}
'{ bus_stmts '}'
{ lex_prev_state();
  if (lib_pin != lib_cell->pins) {
    lib_pin->next = lib_cell->pins;
    lib_cell->pins = lib_pin;
  }
  lib_pin = NULL;
}
semicolon.
;

bus_stmts         : bus_stmt
| bus_stmts bus_stmt
;

bus_stmt          : pin_simple_attr
| BUS_TYPE ':' string semicolon.
{ libc_cell_find_pin_type($3.string); }
| memory_read_group
| memory_write_group
| pin_group_stmt
| pin_group           /* *** limit one pin group only in
this parser */
{ lib_pin = lib_bus;
  lib_bus = NULL;
}
;

memory_read_group : MEM_READ '(' ')' '{'
  ADDRESS ':' name semicolon.
'}
```

libc_yacc.y

```
{ lex_prev_state();
    lib_pin->address_of_memory_read = $7.string;
}
semicolon.
;

memory_write_group : MEM_WRITE '(' ')'
{ lib_memory_write = new_libc_memory_write_rec(); }
'{' mem_write_stmts '}''
{ lex_prev_state();
    free_libc_memory_write_rec(lib_pin->memory_write);
    lib_pin->memory_write = lib_memory_write;
    lib_memory_write = NULL;
}
semicolon.
;

mem_write_stmts : mem_write_stmt
| mem_write_stmts mem_write_stmt
;

mem_write_stmt : ADDRESS ':' name semicolon.
{ lib_memory_write->address = $3.string; }
| CLOCK_ON ':' name semicolon.
{ lib_memory_write->clock_on = $3.string; }
| ENABLE ':' name semicolon.
{ lib_memory_write->enable = $3.string; }
;

/* ----- */

ff_group : C_FF '(' name ',' name ')'
{ lib_ff_latch = new_libc_ff_latch_rec();
    lib_ff_latch->Q_name = $3.string;
    lib_ff_latch->QN_name = $5.string;
    lib_ff_latch->width = 1;
    lib_ff_latch->is_ff = 1;
    lib_ff_latch->is_state = 0;
    lib_ff_latch->clear_preset_var1 = 'X';
    lib_ff_latch->clear_preset_var2 = 'X';
}
'{' ff_stmts '}'
{ lex_prev_state();
/* ---- keep all version */
if (lib_cell != NULL) {
    lib_ff_latch->next = lib_cell->ff_latch;
    lib_cell->ff_latch = lib_ff_latch;
}
else /* ---- inside test_cell */
    free_libc_ff_latch_rec(lib_ff_latch);
    lib_ff_latch = NULL;
}
semicolon.
;

ff_bank_group : C_FFB '(' name ',' name ',' L_INT ')'
{ lib_ff_latch = new_libc_ff_latch_rec(); }
```

```

lib_ff_latch->Q_name = $3.string;
lib_ff_latch->QN_name = $5.string;
lib_ff_latch->width = $7.int_val;
lib_ff_latch->is_ff = 1;
lib_ff_latch->is_state = 0;
lib_ff_latch->clear_preset_var1 = 'X';
lib_ff_latch->clear_preset_var2 = 'X';
}
'{ ff_stmts }'
{ lex_prev_state();
/* ---- keep all version */
if (lib_cell != NULL) {
    lib_ff_latch->next = lib_cell->ff_latch;
    lib_cell->ff_latch = lib_ff_latch;
}
else /* ---- inside test_cell */
    free_libc_ff_latch_rec(lib_ff_latch);
lib_ff_latch = NULL;
}
semicolon.
;

ff_stmts : ff_stmt
| ff_stmts ff_stmt
;

ff_stmt : CLOCK_ON ':' bool_expression semicolon.
{ lib_ff_latch->clock_on = $3.bool_opr; }
| NEXT_ST ':' bool_expression semicolon.
{ lib_ff_latch->next_state = $3.bool_opr; }
| CLEAR ':' bool_expression semicolon.
{ lib_ff_latch->clear = $3.bool_opr; }
| PRESET ':' bool_expression semicolon.
{ lib_ff_latch->preset = $3.bool_opr; }
| CL_PS_V1 ':' var_id
{ libc_cell_preset_var(&(lib_ff_latch->clear_preset_var1),$3.string); }
semicolon.
| CL_PS_V2 ':' var_id
{ libc_cell_preset_var(&(lib_ff_latch->clear_preset_var2),$3.string); }
semicolon.
| ON_ALSO ':' bool_expression semicolon.
{ lib_ff_latch->on_also = $3.bool_opr; /* clock_on_also */ }
}

var_id : identifier
| ''' identifier ''' { $$ = $2; }
;

/* ----- */

latch_group : C_LATCH '(' name ',' name ')'
{ lib_ff_latch = new_libc_ff_latch_rec();
lib_ff_latch->Q_name = $3.string;
lib_ff_latch->QN_name = $5.string;
}
;
```

```

    lib_ff_latch->width    = 1;
    lib_ff_latch->is_ff    = 0;
    lib_ff_latch->is_state = 0;
    lib_ff_latch->clear_preset_var1 = 'X';
    lib_ff_latch->clear_preset_var2 = 'X';
}
{' latch_stmts '}
{ lex_prev_state();
/* ---- keep all version */
if (lib_cell != NULL) {
    lib_ff_latch->next = lib_cell->ff_latch;
    lib_cell->ff_latch = lib_ff_latch;
}
else /* ---- inside test_cell */
    free_libc_ff_latch_rec(lib_ff_latch);
lib_ff_latch = NULL;
}
semicolon.
;

latch_bank_group : C_LB '(' name ',' name ',' L_INT ')'
{ lib_ff_latch = new_libc_ff_latch_rec();
lib_ff_latch->Q_name = $3.string;
lib_ff_latch->QN_name = $5.string;
lib_ff_latch->width = $7.int_val;
lib_ff_latch->is_ff = 0;
lib_ff_latch->is_state = 0;
lib_ff_latch->clear_preset_var1 = 'X';
lib_ff_latch->clear_preset_var2 = 'X';
}
{' latch_stmts '}
{ lex_prev_state();
/* ---- keep all version */
if (lib_cell != NULL) {
    lib_ff_latch->next = lib_cell->ff_latch;
    lib_cell->ff_latch = lib_ff_latch;
}
else /* ---- inside test_cell */
    free_libc_ff_latch_rec(lib_ff_latch);
lib_ff_latch = NULL;
}
semicolon.
;

latch_stmts      : latch_stmt
| latch_stmts latch_stmt
;

latch_stmt       : ENABLE '::' bool_expression semicolon.
| DATA_IN '::' bool_expression semicolon.
| CLEAR '::' bool_expression semicolon.
| PRESET '::' bool_expression semicolon.
| CL_PS_V1 '::' var_id
;
```

```

        { libc_cell_preset_var(&(lib_ff_latch-
>clear_preset_var1),$3.string); }
        semicolon.
        | CL_PS_V2 ':' var_id
        { libc_cell_preset_var(&(lib_ff_latch-
>clear_preset_var2),$3.string); }
        semicolon.
        | ON_ALSO ':' bool_expression semicolon.
        { lib_ff_latch->on_also = $3.bool_opr; /* enable_on_also
*/ }
;

/* ----- */

/* ---- lut group (ignored) */
lut_group      : C_LUT '(' name ')' '{'
                LUT_IP ':' bool_expression semicolon.
                '}'
                { lex_prev_state();
                  free_text_buffer($3.string);
                  free_libc_bool_opr_rec($8.bool_opr);
                }
                semicolon.
;

/* ----- */

/* ---- old method for ff and latch */
state_group    : C_STATE '(' name ',' name ')'
                { lib_ff_latch = new_libc_ff_latch_rec();
                  lib_ff_latch->Q_name = $3.string;
                  lib_ff_latch->QN_name = $5.string;
                  lib_ff_latch->width = 1;
                  lib_ff_latch->is_ff = 0;
                  lib_ff_latch->is_state = 1;
                  lib_ff_latch->clear_preset_var1 = 'X';
                  lib_ff_latch->clear_preset_var2 = 'X';
                }
                '{' state_stmts '}''
                { lex_prev_state();
                  /* ---- keep all version */
                  lib_ff_latch->next = lib_cell->ff_latch;
                  lib_cell->ff_latch = lib_ff_latch;
                  lib_ff_latch = NULL;
                }
                semicolon.
;

state_stmts    : state_stmt
| state_stmts state_stmt
;

state_stmt     : CLOCK_ON ':' bool_expression semicolon.
{ lib_ff_latch->clock_on = $3.bool_opr; }
| NEXT_ST ':' bool_expression semicolon.
{ lib_ff_latch->next_state = $3.bool_opr; }
| ON_ALSO ':' bool_expression semicolon.
;
```

```

enable_on_also */ }
| CLEAR ':' bool_expression semicolon.
| lib_ff_latch->clear = $3.bool_opr; }
| PRESET ':' bool_expression semicolon.
| lib_ff_latch->preset = $3.bool_opr; }
| CL_PS_V1 ':' var_id
| libc_cell_preset_var(&(lib_ff_latch-
>clear_preset_var1),$3.string); }
    semicolon.
| CL_PS_V2 ':' var_id
| libc_cell_preset_var(&(lib_ff_latch-
>clear_preset_var2),$3.string); }
    semicolon.
| ENABLE ':' bool_expression semicolon.
| lib_ff_latch->enable = $3.bool_opr; }
| DATA_IN ':' bool_expression semicolon.
| lib_ff_latch->data_in = $3.bool_opr; }
| FORCE_01 ':' bool_expression semicolon.
| lib_ff_latch->force_01 = $3.bool_opr; }
| FORCE_10 ':' bool_expression semicolon.
| lib_ff_latch->force_10 = $3.bool_opr; }
| FORCE_00 ':' bool_expression semicolon.
| lib_ff_latch->force_00 = $3.bool_opr; }
| FORCE_11 ':' bool_expression semicolon.
| lib_ff_latch->force_11 = $3.bool_opr; }
;

/* ----- */

internal_power_group_c : C_IP '(' name
    { lib_int_power = new_libc_internal_power();
      lib_timing_tbl = new_libc_table_val_rec();
      if (!libc_time_find_template($3.string,tech_lib-
>plut_template,lib_timing_tbl)) {
          libcerror("template not found!");
          exit(1);
      }
      lib_int_power->power = lib_timing_tbl;
      lib_timing_tbl = NULL;
    }
    ')' '{' ipc_stmts '}'
{ lex_prev_state();
  lib_int_power->next      = lib_cell->internal_power_c;
  lib_cell->internal_power_c = lib_int_power;
  lib_int_power = NULL;
}
semicolon.
;

ipc_stmts : ipc_stmt
| ipc_stmts ipc_stmt
;

ipc_stmt : REL_INP ':' name semicolon.
{ lib_int_power->inputs = new_libc_name_list_rec();
  lib_int_power->inputs->name = $3.string;
}
;
```

```

        }
| REL_INPS ':: qname_list semicolon.
{ lib_int_power->inputs = $3.name_list.head; }
| REL_OUTP ':: qname_list semicolon.
{ lib_int_power->outputs = $3.name_list.head; }
| TBL_IDX1 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lib_int_power->power->index1);
  lib_int_power->power->index1 =
libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX2 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lib_int_power->power->index2);
  lib_int_power->power->index2 =
libc_util_float_list2buffer($3.float_list.head); }
| TBL_IDX3 '(' qvalue_list ')' semicolon.
{ free_float_buffer(lib_int_power->power->index3);
  lib_int_power->power->index3 =
libc_util_float_list2buffer($3.float_list.head); }
| VALUES '(' qvalue_lists ')' semicolon.
{ libc_time_values_handle(lib_int_power-
>power,$3.float_list_list.head); }
;

/* ----- */

internal_power_group_p : PIN_IPO '(' ')
{ lib_int_power = new_libc_internal_power(); }
| '{ ip_stmts }'
{ lex_prev_state();
  lib_int_power->next      = lib_pin->internal_power;
  lib_pin->internal_power = lib_int_power;
  lib_int_power = NULL;
}
semicolon.
;

ip_stmts : ip_stmt
| ip_stmts ip_stmt
;

ip_stmt : TI_RP ':: qname_list semicolon.
{ lib_int_power->related_pin = $3.name_list.head; }
| TI_WHEN ':: bool_expression semicolon.
{ lib_int_power->when = $3.bool_opr; }
| IP_EOOO ':: qname_list semicolon.
{ lib_int_power->equal_or_opposite_output =
$3.name_list.head; }
| IP_PL ':: string semicolon.
{ lib_int_power->power_level = $3.string; }
| ip_group_stmt
;

ip_group_stmt : power_group_head '(' name
{ lib_timing_tbl = new_libc_table_val_rec();
  if (!libc_time_find_template($3.string,tech_lib-
>plut_template,lib_timing_tbl)) {
    libcerror("template not found!");
    exit(1);
}
;
```

```

        }
    }
'')' '{' v_stmts '}'
{ lex_prev_state();
  libc_time_power_handle($1.int_val);
}
semicolon.
;

power_group_head : IP_FP      { $$.int_val = 0; }
| IP_RP       { $$.int_val = 1; }
| IP_POWER    { $$.int_val = 2; }
;

/* ----- */

leakage_power_group : C_LPG '(' '
{ lib_lkg_power = new_libc_leakage_power(); }
'{' lp_stmts '}'
{ lex_prev_state();
  lib_lkg_power->next      = lib_cell->leakage_power;
  lib_cell->leakage_power = lib_lkg_power;
  lib_lkg_power = NULL;
}
semicolon.
;

lp_stmts : lp_stmt
| lp_stmts lp_stmt
;

lp_stmt : TI_WHEN ':' bool_expression semicolon.
{ lib_lkg_power->when = $3.bool_opr; }
| VALUE ':' value semicolon.
{ lib_lkg_power->value = $3.real_val; }
;

/* ----- */

memory_group : C_MEM '(' ')
{ lib_memory = new_libc_memory_rec(); }
'{' mem_stmts '}'
{ lex_prev_state();
  free_libc_memory_rec(lib_cell->memory);
  lib_cell->memory = lib_memory;
  lib_memory = NULL;
}
semicolon.
;

mem_stmts : mem_stmt
| mem_stmts mem_stmt
;

mem_stmt : CM_TYPE ':' ram_rom semicolon.
| CM_ADDR_WIDTH ':' L_INT semicolon.
{ lib_memory->address_width = $3.int_val; }
;
```

libc_yacc.y

```
| CM_WORD_WIDTH ':' L_INT semicolon.
| lib_memory->word_width = $3.int_val; }
| CM_C_ADDR ':' "" index_ranges "" semicolon.      /*
ignored */
| CM_R_ADDR ':' "" index_ranges "" semicolon.      /*
ignored */
;

ram_rom      : CM_RAM     { lib_memory->is_ram = 1; }
| CM_ROM     { lib_memory->is_ram = 0; }
;

index_ranges : index_range
| index_ranges index_range
;

index_range   : L_INT
| L_INT ':' L_INT
;

/* ----- */

pin_group    : C_PIN '(' name_list ')'
{ lex_prev_state();
  libc_cell_pin_init($3.name_list.head);
}
'{ pin_stmts }'
{ lex_prev_state();
  lib_pin->next = lib_cell->pins;
  lib_cell->pins = lib_pin;
  lib_pin = NULL;
}
semicolon.
;

pin_stmts    : pin_stmt
| pin_stmts pin_stmt
;

pin_stmt     : pin_simple_attr
| pin_group_stmt
| ignored_stmt
;

pin_simple_attr : PIN_CAP ':' value semicolon.
{ lib_pin->capacitance = $3.real_val; }
| PIN_CLK ':' boolean semicolon.
{ lib_pin->clock = $3.int_val; }
| PIN_CGEP ':' boolean semicolon.
{ lib_pin->clock_gate_enable_pin = $3.int_val; }
| PIN_CC ':' name_list semicolon.
{ lib_pin->connection_class = $3.name_list.head; }
| PIN_DIR ':' direction semicolon.
{ libc_cell_pin_default(); }
| PIN_DF ':' stuck_at_type semicolon.
{ lib_pin->dont_false = $3.sa_type; }
| PIN_DC ':' value semicolon.
```

libc_yacc.y

```
{ lib_pin->drive_current = $3.real_val; }
| PIN_DT ':' driver_type semicolon.
| PIN_ERBF0 { libc_time_delay_model(CMOS2); } ':' value
semicolon.
| PIN_ERBF1 { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERBR0 { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERBR1 { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERF { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERR { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERLF { libc_time_delay_model(CMOS2); } ':' value
| PIN_ERLR { libc_time_delay_model(CMOS2); } ':' value
semicolon.
| PIN_EC ':' L_INT semicolon.
{ lib_pin->emitter_count = $3.int_val; }
| PIN_FCSAT ':' value semicolon.
{ lib_pin->fall_current_slop_after_threshold =
$3.real_val; }

$3.real_val; }

$3.real_val; }

| PIN_FCSBT ':' value semicolon.
{ lib_pin->fall_current_slop_before_threshold =
| PIN_FTAT ':' value semicolon.
{ lib_pin->fall_time_after_threshold = $3.real_val; }
| PIN_FTBT ':' value semicolon.
{ lib_pin->fall_time_before_threshold = $3.real_val; }
| PIN_FWE ':' value semicolon.
{ lib_pin->fall_wor_emitter = $3.real_val; }
| PIN_FWI ':' value semicolon.
{ lib_pin->fall_wor_intercept = $3.real_val; }
| PIN_FL ':' value semicolon.
{ lib_pin->fanout_load = $3.real_val; }
| PIN_FUNCTION ':' bool_expression semicolon.
{ lib_pin->function = $3.bool_opr; }
| PIN_H ':' boolean semicolon.
{ lib_pin->hysteresis = $3.int_val; }
| PIN_IM ':' qname_list semicolon.
{ lib_pin->input_map = $3.name_list.head; }
| PIN_ISL ':' string semicolon.
{ lib_pin->input_signal_level = $3.string; }
| PIN_IV ':' string semicolon.
{ lib_pin->input_voltage = $3.string; }
| PIN_IN ':' string semicolon.          /*
old ? */

{ lib_pin->internal_node = $3.string; }
| PIN_IO ':' boolean semicolon.
{ lib_pin->inverted_output = $3.int_val; }
| PIN_IP ':' boolean semicolon.
{ lib_pin->is_pad = $3.int_val; }
| PIN_MAX_FO ':' n_expression semicolon.      /* change
value to exp */

{ lib_pin->max_fanout = $3.real_val; }
```

libc_yacc.y

```
| PIN_MAX_TRANS ':' n_expression semicolon.          /*  
change value to exp */  
| { lib_pin->max_transition = $3.real_val; }  
| PIN_MAX_CAP ':' n_expression semicolon.          /* change  
value to exp */  
| { lib_pin->max_capacitance = $3.real_val; }  
| PIN_MIN_FO ':' n_expression semicolon.  
| { lib_pin->min_fanout = $3.real_val; }  
| PIN_MIN_TRANS ':' n_expression semicolon.  
| { lib_pin->min_transition = $3.real_val; }  
| PIN_MIN_CAP ':' n_expression semicolon.  
| { lib_pin->min_capacitance = $3.real_val; }  
| PIN_MP ':' value semicolon.  
| { libc_minimum_period_rec *mpp,*nmpp;  
  
    nmpp = new_libc_minimum_period_rec();  
    nmpp->constraint = $3.real_val;  
    if (lib_pin->minimum_period == NULL)  
        lib_pin->minimum_period = nmpp;  
    else {  
        for(mpp=lib_pin->minimum_period;mpp->next;mpp=mpp-  
>next);  
        mpp->next = nmpp;  
    }  
}  
| PIN_MPWH ':' value semicolon.  
| { libc_time_minimum_period(1,$3.real_val); }  
| PIN_MPWL ':' value semicolon.  
| { libc_time_minimum_period(0,$3.real_val); }  
| PIN_MPP ':' boolean semicolon.  
| { lib_pin->multicell_pad_pin = $3.int_val; }  
| PIN_MDL ':' boolean semicolon.  
| { lib_pin->multiple_drivers_legal = $3.int_val; }  
| PIN_NST ':' nextstate_type semicolon.  
| PIN_OSL ':' string semicolon.  
| { lib_pin->output_signal_level = $3.string; }  
| PIN_OV ':' string semicolon.  
| { lib_pin->output_voltage = $3.string; }  
| PIN_PFT ':' pin_func_type semicolon.  
| PIN_PP ':' value semicolon.  
| { lib_pin->pin_power = $3.real_val; }  
| PIN_PT ':' int_or_name semicolon.  
| { lib_pin->prefer_tied = $3.int_val; }  
| PIN_PO ':' boolean semicolon.  
| { lib_pin->primary_output = $3.int_val; }  
| PIN_PC ':' value semicolon.  
| { lib_pin->pulling_current = $3.real_val; }  
| PIN_PR ':' value semicolon.  
| { lib_pin->pulling_resistance = $3.real_val; }  
| PIN_RC ':' value semicolon.  
| { lib_pin->reference_capacitance = $3.real_val; }  
| PIN_RCSAT ':' value semicolon.  
| { lib_pin->rise_current_slop_after_threshold =  
$3.real_val; }  
| PIN_RCSBT ':' value semicolon.  
| { lib_pin->rise_current_slop_before_threshold =  
$3.real_val; }
```

```

| PIN_RTAT ':' value semicolon.
| { lib_pin->rise_time_after_threshold = $3.real_val; }
| PIN_RTBT ':' value semicolon.
| { lib_pin->rise_time_before_threshold = $3.real_val; }
| PIN_RWE ':' value semicolon.
| { lib_pin->rise_wor_emitter = $3.real_val; }
| PIN_RWI ':' value semicolon.
| { lib_pin->rise_wor_intercept = $3.real_val; }
| PIN_SC ':' slew_control semicolon.
| PIN_SF ':' bool_expression semicolon.
| { lib_pin->state_function = $3.bool_opr; }
| PIN_TS ':' bool_expression semicolon.
| { lib_pin->three_state = $3.bool_opr; }
| PIN_VN ':' string semicolon.
| { lib_pin->vhdl_name = $3.string; }
| PIN_WC ':' value semicolon.
| { lib_pin->wire_capacitance = $3.real_val; }
| PIN_WCC ':' string semicolon.
| { lib_pin->wired_connection_class = $3.string; }
| PIN_XF ':' bool_expression semicolon.
| { lib_pin->x_function = $3.bool_opr; }
| CTC_ST ':' signal_type semicolon.
| { libc_cell_pin_type_assign(lib_pin-
>pin_name,$3.sig_type); }
| CTC_TOO ':' boolean semicolon.
;

/* ---- DIR */
direction      : DIR_INPUT { lib_pin->direction = INPUT_E; lib_pin-
>pin_type = l_axubPortTypeIn; }
| DIR_OUTPUT     { lib_pin->direction = OUTPUT_E; lib_pin-
>pin_type = l_axubPortTypeOut; }
| DIR_INOUT     { lib_pin->direction = INOUT_E; lib_pin-
>pin_type = l_axubPortTypeIO; }
| DIR_INTERNAL   { lib_pin->direction = INTERNAL_E; }
;

tc_direction    : DIR_INPUT
| DIR_OUTPUT
| DIR_INOUT
| DIR_INTERNAL
;

/* ---- DRT */
driver_type     : dt_class
| ''' dt_class '''
;

dt_class        : DRT_PULL_UP           { lib_pin->drive_type =
PULL_UP; }
| DRT_PULL_DOWN          { lib_pin->drive_type = PULL_DOWN;
}
| DRT_OPEN_DRAIN         { lib_pin->drive_type = OPEN_DRAIN; }
| DRT_OPEN_SOURCE        { lib_pin->drive_type = OPEN_SOURCE; }
| DRT_BUS_HOLD           { lib_pin->drive_type = BUS_HOLD; }
| DRT_RES                { lib_pin->drive_type = RESISTIVE; }
| DRT_RES0               { lib_pin->drive_type = RESISTIVE0; }
;
```

```

| DRT_RES1      { lib_pin->drive_type = RESISTIVE1; }

/* ---- NST */
nextstate_type : N_DATA      { lib_pin->nextstate_type = DATE_E; }
| N_PRESET     { lib_pin->nextstate_type = PRESET_E; }
| N_CLEAR      { lib_pin->nextstate_type = CLEAR_E; }
| N_LOAD       { lib_pin->nextstate_type = LOAD_E; }
| N_SCAN_IN    { lib_pin->nextstate_type = SCAN_IN_E; }
| N_SCAN_ENABLE { lib_pin->nextstate_type = SCAN_ENABLE_E; }

/* ---- PFT */
pin_func_type : CLK_ENABLE   { lib_pin->pin_func_type =
CLOCK_ENABLE_E; }

| ACT_HIGH     { lib_pin->pin_func_type = ACTIVE_HIGH_E; }

| ACT_LOW      { lib_pin->pin_func_type = ACTIVE_LOW_E; }

ACTIVE_RSING_E; }

| ACT_RISING   { lib_pin->pin_func_type = ACTIVE_RISING_E; }

| ACT_FALLING  { lib_pin->pin_func_type = ACTIVE_FALLING_E; }

int_or_name : L_INT
{ if ($1.int_val < 0 || $1.int_val > 1)
  libcerror("only 0 or 1 is allowed.");
  $$int_val = $1.int_val != 0;
}

' "' L_INT          /* only "0" | "1" */
{ if ($2.int_val < 0 || $2.int_val > 1)
  libcerror("only 0 or 1 is allowed.");
}
' ""

{ $$int_val = $2.int_val != 0; }

/* ---- PSC */
slew_control : NONE_SC     { lib_pin->slew_control = NONE_E; }
| LOW_SC       { lib_pin->slew_control = LOW_E; }
| MED_SC      { lib_pin->slew_control = MED_E; }
| HIGH_SC     { lib_pin->slew_control = HIGH_E; }

pin_group_stmt : timing_group
{ $1.timing->next = lib_pin->timing;;
  lib_pin->timing = $1.timing;
}

| min_pluse_group
| min_period_group
| internal_power_group_p

;

/* ----- */

```

libc_yacc.y

```
routing_track_group      : C_RT '(' name ')'
{ lib_routing_track = new_libc_routing_track_rec();
  lib_routing_track->layer_name = $3.string;
}
'{ rt_stmts }'
{ lex_prev_state();
  lib_routing_track->next = lib_cell->routing_track;
  lib_cell->routing_track = lib_routing_track;
  lib_routing_track = NULL;
}
semicolon.
;

rt_stmts      : rt_stmt
| rt_stmts rt_stmt
;

rt_stmt       : TRACKS ':' L_INT semicolon.
{ lib_routing_track->tracks = $3.int_val; }
| TRACK_AREA ':' value semicolon.
{ lib_routing_track->total_track_area = $3.real_val; }
;
/* ----- */

/* ---- ignored */
statetable_group   : C_ST '(' qname_list ',' qname_list ')' '{'
  TABLE ':' L_QSTRING semicolon.
}'
{ lex_prev_state();
  free_libc_name_list_rec($3.name_list.head);
  free_libc_name_list_rec($5.name_list.head);
  free_text_buffer($10.string);
}
semicolon.
;

/* ----- */

/* ---- ignored */
test_cell_group   : C_TC '(' name. ')'
{ lib_tc_cell = lib_cell; lib_cell = NULL; }
'{ tc_stmts }'
{ lex_prev_state();
  free_text_buffer($3.string);
  lib_cell = lib_tc_cell;
}
semicolon.
;

tc_stmts      : tc_stmt
| tc_stmts tc_stmt
;

tc_stmt       : tc_pin_group
| statetable_group
| ff_group
```

```

| ff_bank_group
| latch_group
| latch_bank_group
;

tc_pin_group      : CTC_PIN '(' name_list ')'
{ lib_tc_pin_name = $3.name_list.head;
lex_prev_state();
}
'{ tc_pin_stmts }'
{ free_libc_name_list_rec(lib_tc_pin_name);
lib_tc_pin_name = NULL;
}
;

tc_pin_stmts     : tc_pin_stmt
| tc_pin_stmts tc_pin_stmt
;

tc_pin_stmt      : CTC_DIR ':' tc_direction semicolon.
| CTC_FUNC ':' ''' simple_bexp ''' /* bool_expression */
{ free_libc_bool_opr_rec($4.bool_opr); } semicolon.
| CTC_FUNC ':' simple_bexp ;
{ free_libc_bool_opr_rec($3.bool_opr); }
| CTC_ST ':' signal_type semicolon.
{ libc_cell_pin_type_assign(lib_tc_pin_name,$3.sig_type); }
|
| CTC_TOO ':' boolean semicolon.
| tc_pin_ignored
;

tc_pin_ignored   : string ':'
{ lex_next_QSTR_state(); }
tc_ignored_attr semicolon.          /* ignore */
{ free_text_buffer($1.string); }
;

tc_ignored_attr : value           { lex_prev_state(); }
| string            { lex_prev_state(); }
free_text_buffer($1.string); }
| L_QSTRING         { free_text_buffer($1.string); }
;

signal_type       : sig_t_class
| ''' sig_t_class '''           { $$ = $2; }
;

sig_t_class       : ST_TSI      { $$.sig_type = ST_TSI_E; }
| ST_TSII     { $$.sig_type = ST_TSII_E; }
| ST_TSO      { $$.sig_type = ST_TSO_E; }
| ST_TSOI     { $$.sig_type = ST_TSOI_E; }
| ST_TSE      { $$.sig_type = ST_TSE_E; }
| ST_TSEI     { $$.sig_type = ST_TSEI_E; }
| ST_TSC      { $$.sig_type = ST_TSC_E; }
| ST_TSCA     { $$.sig_type = ST_TSCA_E; }
| ST_TSCB     { $$.sig_type = ST_TSCB_E; }
;
```

```

| ST_TCLK    { $$.sig_type = ST_TCLK_E; }
;

/* ===== */
timing_group      : TIMING '(' name. ')'
{ libc_time_init($3.string); }
'{ timing_stmts '}'
{ lex_prev_state();
  libc_time_finish();
}
semicolon.
{ $$.timing = lib_timing;
  lib_timing = NULL;
}
;

timing_stmts     : timing_stmt
| timing_stmts timing_stmt
;

timing_stmt       : timing_simple_attr
| timing_complex_attr
| timing_group_stmt
;

timing_simple_attr : TI_ERSFO ':' value semicolon.
{ lib_timing->edge_rate_sensitivity_f0 = $3.real_val; }
| TI_ERSF1 ':' value semicolon.
{ lib_timing->edge_rate_sensitivity_f1 = $3.real_val; }
| TI_ERSR0 ':' value semicolon.
{ lib_timing->edge_rate_sensitivity_r0 = $3.real_val; }
| TI_ERSR1 ':' value semicolon.
{ lib_timing->edge_rate_sensitivity_r1 = $3.real_val; }
| TI_FR ':' value semicolon.
{ lib_timing->fall_resistance = $3.real_val; }
| TI_RR ':' value semicolon.
{ lib_timing->rise_resistance = $3.real_val; }
| TI_IF ':' n_expression semicolon.
{ lib_timing->intrinsic_fall = $3.real_val; }
| TI_IR ':' n_expression semicolon.
{ lib_timing->intrinsic_rise = $3.real_val; }
| TI_RBP ':' bool_expression semicolon.
{ lib_timing->related_bus_pins = $3.bool_opr; }
| TI_ROP ':' related_out_pin semicolon.
| TI_RP ':' qname_list semicolon.
{ libc_cell_relative_pin_handle($3.name_list.head);
  lib_timing->related_pin = $3.name_list.head;
}
| TI_SDF_C ':' { lex_next_SKIP_state(); } sdf_skips
semicolon.
| TI_SDF_E ':' sdf_edge semicolon.
| TI_SF ':' value semicolon.
{ lib_timing->slope_fall = $3.real_val; }
| TI_SR ':' value semicolon.
{ lib_timing->slope_rise = $3.real_val; }
| TI_TT ':' timing_type semicolon.
;
```

```

| TI_TS '::' timing_sense semicolon.
| TI_WHEN '::' bool_expression semicolon.
{ lib_timing->when = $3.bool_opr; }
| TI_WS '::' bool_expression semicolon.
{ lib_timing->when_start = $3.bool_opr; }
| TI_WE '::' bool_expression semicolon.
{ lib_timing->when_end = $3.bool_opr; }
| TI_SDF_CS '::' { lex_next_SKIP_state(); } sdf_skips
semicolon.
| TI_SDF_CE '::' { lex_next_SKIP_state(); } sdf_skips
semicolon.
;

/* ---- SDF_E */
sdf_edge : NO_EDGE
| BOTH_EDGES
| START_EDGE
| END_EDGE
;

/* ---- ROP */
related_out_pin : LTT_ROTONC
{ lib_timing->related_output_pin =
RELATED_OUT_TOTAL_OUTPUT_NET_CAP; }
| LTT_ROONL
{ lib_timing->related_output_pin =
RELATED_OUT_OUTPUT_NET_LENGTH; }
| LTT_ROONWC
{ lib_timing->related_output_pin =
RELATED_OUT_OUTPUT_NET_WIRE_CAP; }
| LTT_ROONPC
{ lib_timing->related_output_pin =
RELATED_OUT_OUTPUT_NET_PIN_CAP; }
;

/* ---- TIT */
timing_type : tt_class
| ''' tt_class '''
;

tt_class : TIT_RE { lib_timing->timing_type =
RISING_EDGE_T; }
| TIT_FE { lib_timing->timing_type = FALLING_EDGE_T; }
| TIT_PS { lib_timing->timing_type = PRESET_T; }
| TIT_CL { lib_timing->timing_type = CLEAR_T; }
| TIT_HR { lib_timing->timing_type = HOLD_RISING_T; }
| TIT_HF { lib_timing->timing_type = HOLD_FALLING_T; }
| TIT_SR { lib_timing->timing_type = SETUP_RISING_T; }
| TIT_SF { lib_timing->timing_type = SETUP_FALLING_T; }
| TIT_RR { lib_timing->timing_type = RECOVERY_RISING_T;
}
| TIT_RF { lib_timing->timing_type = RECOVERY_FALLING_T;
}
| TIT_TSD { lib_timing->timing_type =
THREE_STATE_DISABLE_T; }
| TIT_TSE { lib_timing->timing_type =
THREE_STATE_ENABLE_T; }
;
```

```

        | TIT_RMR { lib_timing->timing_type = REMOVAL_RISING_T; }
        | TIT_RMF { lib_timing->timing_type = REMOVAL_FALLING_T;
    }

    COMBINATIONAL_T; } | TIT_C { lib_timing->timing_type =
        | TIT_SKR { lib_timing->timing_type = SKEW_RISING_T; }
        | TIT_SKF { lib_timing->timing_type = SKEW_FALLING_T; }
        | TIT_NSHR { lib_timing->timing_type =
    NON_SEQ_HOLD_RISING_T; }
        | TIT_NSHF { lib_timing->timing_type =
    NON_SEQ_HOLD_FALLING_T; }
        | TIT_NSSR { lib_timing->timing_type =
    NON_SEQ_SETUP_RISING_T; }
        | TIT_NSSF { lib_timing->timing_type =
    NON_SEQ_SETUP_FALLING_T; }
        | TIT_NCHH { lib_timing->timing_type =
    NOCHANGE_HIGH_HIGH_T; }
        | TIT_NCHL { lib_timing->timing_type =
    NOCHANGE_HIGH_LOW_T; }
        | TIT_NCLH { lib_timing->timing_type =
    NOCHANGE_LOW_HIGH_T; }
        | TIT_NCLL { lib_timing->timing_type = NOCHANGE_LOW_LOW_T;
    }

;

/* ---- TST */
timing_sense : ts_type
    | ''' ts_type '''
;

ts_type : TIS_POS { lib_timing->timing_sense = POSITIVE_UNATE_E;
}
    | TIS_NEG { lib_timing->timing_sense = NEGATIVE_UNATE_E;
}
    | TIS_NON { lib_timing->timing_sense = NON_UNATE_E; }

;

timing_complex_attr : TI_FDI { libc_time_delay_model(PIECEWISE_CMOS); }
'(' L_INT ',' value ')'
    | semicolon.
        { libc_time_piecewise($4.int_val,1,$6.real_val); }
    | TI_FNT { libc_time_delay_model(PIECEWISE_CMOS); } ')'
L_INT ',' value ')
    | semicolon.
        { libc_time_piecewise($4.int_val,2,$6.real_val); }
    | TI_FPR { libc_time_delay_model(PIECEWISE_CMOS); } ')'
L_INT ',' value ')
    | semicolon.
        { libc_time_piecewise($4.int_val,3,$6.real_val); }
    | TI_FWR { libc_time_delay_model(PIECEWISE_CMOS); } ')'
L_INT ',' value ')
    | semicolon.
        { libc_time_piecewise($4.int_val,4,$6.real_val); }
    | TI_RDI { libc_time_delay_model(PIECEWISE_CMOS); } ')'
L_INT ',' value ')
    | semicolon.
        { libc_time_piecewise($4.int_val,5,$6.real_val); }
    | TI_RNT { libc_time_delay_model(PIECEWISE_CMOS); } ')'
L_INT ',' value ')
    | semicolon.
        { libc_time_piecewise($4.int_val,6,$6.real_val); }
;

```

libc_yacc.y

```
| TI_RPR { libc_time_delay_model(PIECEWISE_CMOS); } '(
L_INT ',' value ')' semicolon.
| libc_time_piecewise($4.int_val,7,$6.real_val); }
| TI_RWR { libc_time_delay_model(PIECEWISE_CMOS); } '(
L_INT ',' value ')' semicolon.
| libc_time_piecewise($4.int_val,8,$6.real_val); }
;

/* ----- */

timing_group_stmt : timing_group_head '(' name
    { lib_timing_tbl = new_libc_table_val_rec();
      if (!libc_time_find_template($3.string,tech_lib-
>lut_template,lib_timing_tbl)) {
          libcerror("template not found!");
          exit(1);
      }
    }
    ')' '{' v_stmts '}'
    { lex_prev_state();
      libc_time_handle($1.int_val);
    }
    semicolon.
;

timing_group_head : CELL_DEGR { $$ .int_val = 0;
    libc_time_delay_model(TABLE_LOOKUP);
    | CELL_RISE { $$ .int_val = 1;
    libc_time_delay_model(TABLE_LOOKUP);
    | CELL_FALL { $$ .int_val = 2;
    libc_time_delay_model(TABLE_LOOKUP);
    | R_PROP { $$ .int_val = 3;
    libc_time_delay_model(TABLE_LOOKUP);
    | F_PROP { $$ .int_val = 4;
    libc_time_delay_model(TABLE_LOOKUP);
    | R_TRANS { $$ .int_val = 5;
    libc_time_delay_model(TABLE_LOOKUP);
    | F_TRANS { $$ .int_val = 6;
    libc_time_delay_model(TABLE_LOOKUP);
    | R_CONS { $$ .int_val = 7;
    libc_time_delay_model(TABLE_LOOKUP);
    | F_CONS { $$ .int_val = 8;
    libc_time_delay_model(TABLE_LOOKUP);
    ;
}

v_stmts : v_stmt
| v_stmts v_stmt
;

v_stmt : TBL_IDX1 '(' qvalue_list ')' semicolon.
    { free_float_buffer(lib_timing_tbl->index1);
      lib_timing_tbl->index1 =
    libc_util_float_list2buffer($3.float_list.head); }
    | TBL_IDX2 '(' qvalue_list ')' semicolon.
    { free_float_buffer(lib_timing_tbl->index2);
      lib_timing_tbl->index2 =
    libc_util_float_list2buffer($3.float_list.head); }
```

libc_yacc.y

```
| TBL_IDX3 '(' qvalue_list ')' semicolon.  
| free_float_buffer(lib_timing_tbl->index3);  
| lib_timing_tbl->index3 =  
libc_util_float_list2buffer($3.float_list.head); }  
| VALUES '(' qvalue_lists ')' semicolon.  
{  
libc_time_values_handle(lib_timing_tbl,$3.float_list_list.head); }  
;  
/* ===== */  
  
min_pluse_group : MIN_PLUSE_WIDTH '(' ')'  
{ lib_mpw = new_libc_min_pulse_width_rec(); }  
'{' mpw_stmts '}'  
{ lex_prev_state();  
lib_mpw->next = lib_pin->min_pulse_width;  
lib_pin->min_pulse_width = lib_mpw;  
lib_mpw = NULL;  
}  
semicolon.  
;  
  
mpw_stmts : mpw_stmt  
| mpw_stmts mpw_stmt  
;  
  
mpw_stmt : MPW_CH ':' value semicolon.  
{ lib_mpw->constraint_high = $3.real_val; }  
| MPW_CL ':' value semicolon.  
{ lib_mpw->constraint_low = $3.real_val; }  
| TI_WHEN ':' bool_expression semicolon.  
{ lib_mpw->when = $3.bool_opr; }  
| TI_SDF_C ':' { lex_next_SKIP_state(); } sdf_skips  
semicolon.  
;  
/* ----- */  
  
min_period_group : MIN_PERIOD '(' ')'  
{ lib_mp = new_libc_minimum_period_rec(); }  
'{' mp_stmts '}'  
{ lex_prev_state();  
lib_mp->next = lib_pin->minimum_period;  
lib_pin->minimum_period = lib_mp;  
lib_mp = NULL;  
}  
semicolon.  
;  
  
mp_stmts : mp_stmt  
| mp_stmts mp_stmt  
;  
  
mp_stmt : MP_C ':' value semicolon.  
{ lib_mp->constraint = $3.real_val; }  
| TI_WHEN ':' bool_expression semicolon.  
{ lib_mp->when = $3.bool_opr; }
```

libc_yacc.y

```
| TI_SDF_C ':' { lex_next_SKIP_state(); } sdf_skips
semicolon.
;

/* ===== */
n_expression      : simple_exp
| ''' simple_exp ''' { $$.real_val = $2.real_val; }
;

simple_exp       : term
| simple_exp '+' term
{ $$.real_val = $1.real_val + $3.real_val; }
| simple_exp '-' term
{ $$.real_val = $1.real_val - $3.real_val; }
;

term             : factor
| term '*' factor
{ $$.real_val = $1.real_val * $3.real_val; }
| term '/' factor
{ $$.real_val = $1.real_val / $3.real_val; }
;

factor           : primary
| '+' primary          { $$.real_val = $2.real_val; }
| '-' primary          { $$.real_val = - $2.real_val; }
;

primary          : '(' simple_exp ')'
{ $$.real_val = $2.real_val; }
| identifier
{ if (!libc_def_gc_find($1.string,&($$.real_val))) {
    libcerror("define not found.");
    $$.real_val = 1.0;
}
}
| L_INT
{ $$.real_val = (float) $1.int_val; }
| L_REAL
;

/* ----- */

bool_expression   : simple_bexp
| ''' simple_bexp ''' { $$.bool_opr = $2.bool_opr;
}
;

simple_bexp       : or_exp
| simple_bexp '^' or_exp
{ $$.bool_opr = libc_opr_handle(XOR_B,0);
$$.bool_opr->L = $1.bool_opr;
$$.bool_opr->R = $3.bool_opr;
}
;
```

```

or_exp      : and_exp
| or_exp '||' and_exp
{ $$.bool_opr = libc_opr_handle(OR_B,0);
  $$.bool_opr->L = $1.bool_opr;
  $$.bool_opr->R = $3.bool_opr;
}
| or_exp '+' and_exp
{ $$.bool_opr = libc_opr_handle(OR_B,0);
  $$.bool_opr->L = $1.bool_opr;
  $$.bool_opr->R = $3.bool_opr;
}
;

and_exp    : uni_exp
| and_exp '&' uni_exp
{ $$.bool_opr = libc_opr_handle(AND_B,0);
  $$.bool_opr->L = $1.bool_opr;
  $$.bool_opr->R = $3.bool_opr;
}
| and_exp '*' uni_exp
{ $$.bool_opr = libc_opr_handle(AND_B,0);
  $$.bool_opr->L = $1.bool_opr;
  $$.bool_opr->R = $3.bool_opr;
}
| and_exp uni_exp
{ $$.bool_opr = libc_opr_handle(AND_B,0);
  $$.bool_opr->L = $1.bool_opr;
  $$.bool_opr->R = $2.bool_opr;
}
;

uni_exp    : b_primary
| '!' b_primary
{ $$.bool_opr = libc_opr_handle(NOT_B,0);
  $$.bool_opr->R = $2.bool_opr;
}
| b_primary ''
{ $$.bool_opr = libc_opr_handle(NOT_B,0);
  $$.bool_opr->R = $1.bool_opr;
}
;

b_primary  : '(' simple_bexp ')'
{ $$.bool_opr = $2.bool_opr; }
| id_name
| L_INT           /* only 0 and 1 is allowed */
{ if ($1.int_val < 0 || $1.int_val > 1)
    libcerror("only 0 or 1 is allowed in boolean
expression.");
    if ($1.int_val)
      $$.bool_opr = libc_opr_handle(ONE_B,1);
    else
      $$.bool_opr = libc_opr_handle(ZERO_B,0);
}
;

id_name    : identifier

```

libc_yacc.y

```
{ $$._bool_opr = libc_opr_handle(ID_B,(int)$1.string); }
| identifier '[' simple_exp ']'
{ $$._bool_opr = libc_opr_handle(INDEX_B,0);
  $$._bool_opr->L = libc_opr_handle(ID_B,(int)$1.string);
  $$._bool_opr->R =
  libc_opr_handle(CONST_B,(int)$3.real_val);
}
;

/* ===== */
%%

/* #include "lex.yy.c" */

/* ----- */

public
int parse_a_tlib_file(
    char *f_name)
{ int code = 0;

    file_name = f_name;
    libcin = fopen(f_name,"r");
    if (libcin == NULL) {
        fprintf(stderr,"Could not open input file %s.\n",f_name);
        return(1);
    }
    code = yyparse();
    fclose(libcin);
    return(code);
}

/* ----- */

#if 0
int main(
    int argc,
    char **argv)
{
    file_name = argv[1];
    libcin = fopen(argv[1],"r");
    yyparse();
    fclose(libcin);
}
#endif
```

libc_libcds.h

```
/*=====
=====
struct any_unit_rec {
    float V;
    float unit;
};
typedef struct any_unit_rec any_unit_rec;

/* -----
 */

struct libc_name_list_rec /*COPY*/ {
    text_buffer             *name;
    struct libc_name_list_rec *next;
};
typedef struct libc_name_list_rec libc_name_list_rec;

/* -----
 */

struct libc_name_list_list {
    struct libc_name_list_rec *name_list1;
    struct libc_name_list_rec *name_list2;
    struct libc_name_list_list *next;
};
typedef struct libc_name_list_list libc_name_list_list;

/* -----
 */

struct libc_float_list_rec {
    float fvalue;
    struct libc_float_list_rec *next;
};
typedef struct libc_float_list_rec libc_float_list_rec;

/* -----
 */

struct libc_float_list_list {
    struct libc_float_list_rec *v_list;
    struct libc_float_list_list *next;
};
typedef struct libc_float_list_list libc_float_list_list;

/* -----
*/
/* ----- */

struct libc_define_rec {
    text_buffer *name;
    text_buffer *object;
    text_buffer *type;
    struct libc_define_rec *next;
};
typedef struct libc_define_rec libc_define_rec;

/* ----- */
```

libc_libcds.h

```
enum pad_type_E { PAD_SLOTS, PAD_DRIVER_SITES, PAD_INPUT_DRIVER_SITES,
PAD_OUTPUT_DRIVER_SITES };
typedef enum pad_type_E pad_type_E;

struct libc_cell_area_rec {
    text_buffer *area_name;
    enum pad_type_E resource_type;
    struct libc_cell_area_rec *next;
};
typedef struct libc_cell_area_rec libc_cell_area_rec;

/* ----- */
/* ===== */
/* ---- scaled_factor_group */

struct libc_k_factor_rec {
    text_buffer *kf_name;

    /* ---- scaling attr 0: process, 1:temp, 2:volt */
    float k_cell_rise[3];
    float k_cell_fall[3];
    float k_cell_leakage_power[3];
    float k_cell_power[3];
    float k_drive_current[3];
    float k_drive_fall[3];
    float k_drive_rise[3];
    float k_fall_delay_intercept[3];
    float k_fall_pin_resistance[3];
    float k_fall_propagation[3];
    float k_fall_transition[3];
    float k_fall_wire_resistance[3];
    float k_fall_wor_emitter[3];
    float k_fall_wor_intercept[3];
    float k_hold_fall[3];
    float k_hold_rise[3];
    float k_internal_power[3];
    float k_intrinsic_fall[3];
    float k_intrinsic_rise[3];
    float k_min_period[3];
    float k_min_pulse_width_high[3];
    float k_min_pulse_width_low[3];
    float k_nochange_fall[3];
    float k_nochange_rise[3];
    float k_pin_cap[3];
    float k_pin_power[3];
    float k_recovery_fall[3];
    float k_recovery_rise[3];
    float k_removal_fall[3];
    float k_removal_rise[3];
    float k_rise_delay_intercept[3];
    float k_rise_pin_resistance[3];
    float k_rise_propagation[3];
    float k_rise_transition[3];
    float k_rise_wire_resistance[3];
    float k_rise_wor_emitter[3];
```

libc_libcds.h

```
float          k_rise_wor_intercept[3];
float          k_setup_fall[3];
float          k_setup_rise[3];
float          k_skew_fall[3];
float          k_skew_rise[3];
float          k_slope_fall[3];
float          k_slope_rise[3];
float          k_wire_cap[3];
float          k_wire_res[3];

struct libc_k_factor_rec *next;
};

typedef struct libc_k_factor_rec libc_k_factor_rec;

/* ===== */

enum delay_model_E { UNKNOW_M, GENERIC_ECL, GENERIC_CMOS, TABLE_LOOKUP,
CMOS2, PIECEWISE_CMOS };
typedef enum delay_model_E delay_model_E;

enum in_place_swap_mode_E { MATCH_FOOTPRINT, NO_SWAPPING, IGNORE_FOOTPRINT };
typedef enum in_place_swap_mode_E in_place_swap_mode_E;

enum mmp_E { MAX_E, MIN_E, PLUS_E };      /* nonpaired_twin_inc_delay_func */
typedef enum mmp_E mmp_E;

enum piece_type_E { PIECE_LENGTH, PIECE_WIRE_CAP, PIECE_PIN_CAP,
PIECE_TOTAL_CAP };
typedef enum piece_type_E piece_type_E;

enum wired_logic_function_E { WIRED_AND, WIRED_OR };
typedef enum wired_logic_function_E wired_logic_function_E;

enum default_wire_load_mode_E { SEGMENTED_WL, TOP_WL, ENCLOSED_WL };
typedef enum default_wire_load_mode_E default_wire_load_mode_E;

/* ----- */

struct libc_lib_rec {
    text_buffer           *lib_name;

    /* ---- simple attr */
    text_buffer           *aux_no_pulldown_pin_property;
    text_buffer           *bus_naming_style;
    text_buffer           *comment;
    struct any_unit_rec   *current_unit;
    text_buffer           *date;
    enum delay_model_E    delay_model;
    enum in_place_swap_mode_E in_place_swap_mode;
    struct any_unit_rec   *leakage_power_unit;
    int                  max_wired_emitters;
    int                  multiple_drivers_legal;      /* boolean */
    float                nom_process;
    float                nom_temperature;
    float                nom_voltage;
    enum mmp_E            nonpaired_twin_inc_delay_func;
    text_buffer           *no_pulldown_pin_property;
```

libc_libcds.h

```
enum piece_type_E          piece_type;
struct any_unit_rec        *power_unit;
text_buffer                 *preferred_output_pad_slew_rate_control;
text_buffer                 *preferred_output_pad_voltage;
text_buffer                 *preferred_input_pad_voltage;
struct any_unit_rec        *pulling_resistance_unit;
float                      reference_capacitance;           /* cmos2 */
text_buffer                 *revision;
int                        simulation;                  /* boolean */
struct any_unit_rec        *time_unit;
text_buffer                 *unconnected_pin_property;
struct any_unit_rec        *voltage_unit;
enum wired_logic_function_E wired_logic_function;

/* ---- default attr */
float                      default_cell_leakage_power;
float                      default_cell_power;
struct libc_name_list_rec  *default_connection_class;
int                        default_emitter_count;
float                      default_edge_rate_breakpoint_f0;    /* cmos2 */
float                      default_edge_rate_breakpoint_f1;    /* cmos2 */
float                      default_edge_rate_breakpoint_r0;    /* cmos2 */
float                      default_edge_rate_breakpoint_r1;    /* cmos2 */
float                      default_fall_delay_intercept; /* */

piecewise */
float                      default_fall_nonpaired_twin;
float                      default_fall_pin_resistance; /* */

piecewise */
float                      default_fall_wire_resistance;
float                      default_fall_wor_emitter;
float                      default_fall_wor_intercept;
float                      default_fanout_load;
float                      default_inout_pin_cap;
float                      default_inout_pin_fall_res;
float                      default_inout_pin_rise_res;
float                      default_input_pin_cap;
float                      default_intrinsic_fall;
float                      default_intrinsic_rise;
float                      default_max_capacitance;
float                      default_max_fanout;
float                      default_max_transition;
float                      default_max_utilization;
float                      default_min_porosity;
text_buffer                 *default_operating_conditions;
float                      default_output_pin_cap;
float                      default_output_pin_fall_res;
float                      default_output_pin_rise_res;
int                        default_pin_limit;
int                        default_pin_power;
float                      default_rc_fall_coefficient; /* cmos2 */
float                      default_rc_rise_coefficient; /* cmos2 */
float                      default_rise_delay_intercept; /* */

piecewise */
float                      default_rise_nonpaired_twin;
float                      default_rise_pin_resistance; /* */

piecewise */
float                      default_rise_wire_resistance;
```

libc_libcds.h

```
float           default_rise_wor_emitter;
float           default_rise_wor_intercept;
float           default_setup_coefficient;          /* cmos2 */
float           default_slope_fall;
float           default_slope_rise;
text_buffer     *default_wire_load;
float           default_wire_load_area;
float           default_wire_load_capacitance;
enum default_wire_load_mode_E   default_wire_load_mode;
float           default_wire_load_resistance;
text_buffer     *default_wire_load_selection;

struct libc_k_factor_rec    *k_factor;           /* for library */
struct libc_k_factor_rec    *sc_k_factor;         /* for cell
(scaled_factor_group) */

/* ---- complex attr */
struct any_unit_rec        *capacitive_load_unit;
struct libc_define_rec      *define;
struct libc_cell_area_rec   *define_cell_area;
float_buffer               *piece_define;
struct libc_name_list_rec   *routing_layers;
text_buffer                *technology;

/* ---- lib_group_stmt */

struct libc_lu_table_template_rec *lut_template;
struct libc_lu_table_template_rec *plut_template;
struct libc_operating_condition_rec *operating_cond;
struct libc_power_supply_rec     *power_supply;
struct libc_table_val_rec       *rise_tr_table;
struct libc_table_val_rec       *fall_tr_table;
struct libc_timing_range_rec    *timing_range;
struct libc_type_rec           *type;
struct libc_wire_load_rec      *wire_load;
struct libc_wire_load_selection_rec *wire_load_selection; /* *
wire_load_selection group */

struct libc_cell_rec          *cells;
struct libc_define_value_rec   *def_val;
void                         *hook;
};

typedef struct libc_lib_rec libc_lib_rec;

/* ===== */

enum variable_E {
    VARIABLE_E_NONE,
    INPUT_NET_TRANSITION, TOTAL_OUTPUT_NET_CAPACITANCE, OUTPUT_NET_LENGTH,
    OUTPUT_NET_WIRE_CAP, OUTPUT_NET_PIN_CAP,
    RELATED_OUT_TOTAL_OUTPUT_NET_CAP,
    RELATED_OUT_OUTPUT_NET_LENGTH,
    RELATED_OUT_OUTPUT_NET_WIRE_CAP,
    RELATED_OUT_OUTPUT_NET_PIN_CAP,
    CONSTRAINED_PIN_TRANSITION,
```

libc_libcds.h

```
RELATED_PIN_TRANSITION,
OUTPUT_PIN_TRANSITION, CONNECT_DELAY
};

typedef enum variable_E variable_E;

/* ----- */

struct libc_lu_table_template_rec {
    text_buffer          *tt_name;
    enum variable_E     variable_1;
    enum variable_E     variable_2;
    enum variable_E     variable_3;
    float_buffer        *index_1;
    float_buffer        *index_2;
    float_buffer        *index_3;
    struct libc_lu_table_template_rec *next;
};
typedef struct libc_lu_table_template_rec libc_lu_table_template_rec;

/* ===== */

enum tree_type_E { BEST_CASE_TREE, BALANCED_TREE, WORST_CASE_TREE };
typedef enum tree_type_E tree_type_E;

/* ----- */

struct libc_oc_power_rail_rec {
    text_buffer          *power_supply;
    float                volt;
    struct libc_oc_power_rail_rec *next;
};
typedef struct libc_oc_power_rail_rec libc_oc_power_rail_rec;

/* ----- */

struct libc_operating_condition_rec {
    text_buffer          *oc_name;
    float                process;
    float                temperature;
    float                voltage;
    enum tree_type_E     tree_type;
    struct libc_oc_power_rail_rec *power_rail;
    struct libc_operating_condition_rec *next;
};
typedef struct libc_operating_condition_rec libc_operating_condition_rec;

/* ----- */

struct libc_power_supply_rec {
    text_buffer          *ps_name;
    text_buffer          *default_ps;
    struct libc_oc_power_rail_rec *power_rail;
    struct libc_power_supply_rec *next;
};
typedef struct libc_power_supply_rec libc_power_supply_rec;

/* ===== */
```

libc_libcds.h

```
struct libc_timing_range_rec {
    text_buffer          *tr_name;
    float                faster_factor;
    float                slower_factor;
    struct libc_timing_range_rec *next;
};

typedef struct libc_timing_range_rec libc_timing_range_rec;

/* ===== */

struct libc_type_rec {
    text_buffer          *type_name;
    int                  bit_from;           /* default is 0 */
    int                  bit_to;             /* default is 0 */
    int                  bit_width;          /* default is 1 */
    int                  downto;             /* default is false boolean
*/;
    struct libc_type_rec *next;
};

typedef struct libc_type_rec libc_type_rec;

/* ===== */

struct libc_fanout_length_rec {
    int                  fanout;
    float               length;
    float               capacitance;
    float               resistance;
    float               area;

    float               ave_cap;            /* ---- for floorplan manager */
    float               std_dev;             /* ---- for floorplan manager */
    int                  no_of_net;          /* ---- for floorplan manager
*/;
    struct libc_fanout_length_rec *next;
};

typedef struct libc_fanout_length_rec libc_fanout_length_rec;

/* ----- */

/* ---- wire_load and wire_load_table group */

struct libc_wire_load_rec {
    text_buffer          *wl_name;
    float                area;
    float                capacitance;
    float                resistance;
    float                slope;
    struct libc_fanout_length_rec *fanout_length;

    struct libc_wire_load_rec *next;
};

typedef struct libc_wire_load_rec libc_wire_load_rec;

/* ===== */
```

libc_libcds.h

```
/* ---- (for wire_load_selection group) sort by area */

struct libc_wire_load_from_area_rec {
    float min_area;
    float max_area;
    struct libc_wire_load_rec *_load_model;
    struct libc_wire_load_from_area_rec *next;
};

typedef struct libc_wire_load_from_area_rec libc_wire_load_from_area_rec;

/* ----- */

struct libc_wire_load_selection_rec {
    text_buffer *wls_name;
    struct libc_wire_load_from_area_rec *area_table;
    struct libc_wire_load_selection_rec *next;
};

typedef struct libc_wire_load_selection_rec libc_wire_load_selection_rec;

/* ===== */
/* ===== */

enum dont_false_E { SA0, SA1, SA01 };
typedef enum dont_false_E dont_false_E;

/* ----- */

/* ---- cell, test_cell group */

struct libc_cell_rec {
    struct libc_lib_rec *_tlib;
    text_buffer *cell_name;

    /* ---- simple attr */
    float area;
    int auxiliary_pad_cell; /* boolean */
    text_buffer *cell_footprint;
    float cell_power;
    float cell_leakage_power;
    struct libc_bool_opr_rec *contention_condition;
    enum dont_false_E dont_false;
    int dont_touch; /* boolean */
    int dont_use; /* boolean */
    text_buffer *geometry_print; /* bus, bundle cells
only */
    int handle_negative_constraint; /* boolean */
    int interface_timing; /* boolean */
    int map_only; /* boolean */
    int pad_cell; /* boolean */
    int pad_type; /* clock only */
    int pin_limit;
    int preferred; /* boolean */
    struct libc_k_factor_rec *_scaling_factors;
    text_buffer *scan_group; /* test-cell only */
    text_buffer *single_bit_degenerate; /* block-box,
bus, bundle cells only */
}
```

libc_libcds.h

```
text_buffer           *vhdl_name;

/* ---- complex attr */
struct libc_name_list_list  *pin_equal;
struct libc_name_list_list  *pin_opposite;

/* ---- cell group stmt */
/* bundle, bus, pin,statetable,test_cell ??? */

/* ---- bundle, bus, pin group */
struct libc_pin_rec *pins;

/* ---- internal_power group */
struct libc_internal_power  *internal_power_c;
struct libc_leakage_power   *leakage_power;

/* ---- for ff, ff_bank, latch, or latch_bank group */
struct libc_ff_latch_rec   *ff_latch;

/* ---- memory group */
struct libc_memory_rec *memory;

/* ---- routing_track group */
struct libc_routing_track_rec  *routing_track;

struct libc_define_value_rec      *def_val;

struct libc_cell_rec             *next;

int                           *extra;
void                          *hook;
};

typedef struct libc_cell_rec libc_cell_rec;

/* ===== */

struct libc_memory_write_rec /*COPY*/ {
    text_buffer           *address;
    text_buffer           *clock_on;
    text_buffer           *enable;
};
typedef struct libc_memory_write_rec libc_memory_write_rec;

/* ----- */

/* ---- order can not change */
enum libc_bool_type_E {
    XOR_B, OR_B, AND_B, NOT_B, /* ^ | & ! */
    ID_B,                  /* identifier (by name) */
    INDEX_B,               /* index name */
    CONST_B,                /* constant value */
    ZERO_B, ONE_B          /* 0, 1 */
};
typedef enum libc_bool_type_E libc_bool_type_E;

struct libc_bool_opr_rec /*COPY*/ /*(type)*/ {
    enum libc_bool_type_E type;
```

libc_libcds.h

```
union /*(type)*/
{
    text_buffer      /*(ID_B)*/      *id_name;
    int              /*(default)*/     value;
} u;
struct libc_bool_opr_rec   *L;
struct libc_bool_opr_rec   *R;
};

typedef struct libc_bool_opr_rec libc_bool_opr_rec;

/* ----- */

enum direction_E { INPUT_E, OUTPUT_E, INOUT_E, INTERNAL_E };
typedef enum direction_E direction_E;

enum drive_type_E { PULL_UP, PULL_DOWN, OPEN_DRAIN, OPEN_SOURCE, BUS_HOLD,
RESISTIVE, RESISTIVE0, RESISTIVE1 };
typedef enum drive_type_E drive_type_E;

enum nextstate_type_E { DATE_E, PRESET_E, CLEAR_E, LOAD_E, SCAN_IN_E,
SCAN_ENABLE_E };
typedef enum nextstate_type_E nextstate_type_E;

enum pin_func_type_E { CLOCK_ENABLE_E, ACTIVE_HIGH_E, ACTIVE_LOW_E,
ACTIVE_RSING_E, ACTIVE_FALLING_E };
typedef enum pin_func_type_E pin_func_type_E;

enum slew_control_E { NONE_E, LOW_E, MED_E, HIGH_E };
typedef enum slew_control_E slew_control_E;

enum signal_type_E {
    ST_TSI_E, ST_TSII_E, ST_TSO_E, ST_TSOI_E, ST_TSE_E, ST_TSEI_E, ST_TSC_E,
ST_TSCA_E, ST_TSCB_E, ST_TCLK_E };
typedef enum signal_type_E signal_type_E;

/* ----- */

/* ---- bundle,bus,pin group */
struct libc_pin_rec /*COPY*/{

    struct libc_name_list_rec *pin_name;           /* for bundle,bus,pin
group */
    struct libc_cell_rec     *_current_cell;

    struct libc_name_list_rec *members;            /* for bundle group */
    int                      is_bus;                /* for bus */
    struct libc_type_rec     *_bus_type;            /* for bus group */

    /* ---- simple attr */
    float                    capacitance;
    int                      clock;                 /* boolean */
    int                      clock_gate_enable_pin; /* boolean */
    struct libc_name_list_rec *connection_class;
    enum direction_E         direction;
    enum dont_false_E        dont_false;
    float                    drive_current;
    enum drive_type_E        drive_type;
    int                      emitter_count;
}
```

libc_libcds.h

```
float          fall_current_slop_after_threshold;
float          fall_current_slop_before_threshold;
float          fall_time_after_threshold;
float          fall_time_before_threshold;
float          fall_wor_emitter;
float          fall_wor_intercept;
float          fanout_load;
struct libc_bool_opr_rec *function;
int            hysteresis; /* boolean */
struct libc_name_list_rec *input_map;
text_buffer    *input_signal_level;
text_buffer    *input_voltage;
text_buffer    *internal_node; /* old */
int            inverted_output; /* boolean */
int            is_pad; /* boolean */
float          max_fanout;
float          max_transition;
float          max_capacitance;
float          min_fanout;
float          min_transition;
float          min_capacitance;
int            multicell_pad_pin; /* boolean */
int            multiple_drivers_legal; /* boolean */
enum nextstate_type_E nextstate_type;
text_buffer    *output_signal_level;
text_buffer    *output_voltage;
enum pin_func_type_E pin_func_type;
float          pin_power;
int            prefer_tied;
int            primary_output;
float          pulling_current;
float          pulling_resistance;
float          reference_capacitance;
float          rise_current_slop_after_threshold;
float          rise_current_slop_before_threshold;
float          rise_time_after_threshold;
float          rise_time_before_threshold;
float          rise_wor_emitter;
float          rise_wor_intercept;
enum slew_control_E slew_control;
struct libc_bool_opr_rec *state_function;
struct libc_bool_opr_rec *three_state;
text_buffer    *vhdl_name;
float          wire_capacitance;
struct wired_connection_class *wired_connection_class;
struct x_function *x_function;

/* ---- group stmt */
text_buffer    *address_of_memory_read; /* in bus group */
*/
struct libc_memory_write_rec *memory_write; /* in bus group */

struct libc_timing_rec *timing;
struct libc_min_pulse_width_rec *min_pulse_width;
struct libc_minimum_period_rec *minimum_period;
```

libc_libcds.h

```
/* ---- internal_power group */
struct libc_internal_power *internal_power;

struct libc_define_value_rec           *def_val;

/* ---- inside use */
int          pin_type;

struct libc_pin_rec      *next;
};

typedef struct libc_pin_rec libc_pin_rec;

/* ===== */

/* ---- ff, ff_bank, latch, latch_bank (state) group */
struct libc_ff_latch_rec {
    text_buffer          *Q_name;
    text_buffer          *QN_name;
    int                  width;                      /* 1: normal, >1 bank
*/
    int                  is_ff:1;
    int                  is_state:1;                /* old model */
    char                clear_preset_var1;        /* L, H, N, T, X
*/
    char                clear_preset_var2;        /* L, H, N, T, X
*/
    struct libc_bool_opr_rec *clear;
    struct libc_bool_opr_rec *preset;
    struct libc_bool_opr_rec *clock_on;            /* ff group */
    struct libc_bool_opr_rec *next_state;          /* ff group */
    struct libc_bool_opr_rec *on_also;              /* clock_on_also,
enable_on_also */
    struct libc_bool_opr_rec *enable;                /* latch group */
    struct libc_bool_opr_rec *data_in;               /* latch group */
    struct libc_bool_opr_rec *force_00;              /* state group */
    struct libc_bool_opr_rec *force_01;              /* state group */
    struct libc_bool_opr_rec *force_10;              /* state group */
    struct libc_bool_opr_rec *force_11;              /* state group */
    struct libc_ff_latch_rec *next;
};

typedef struct libc_ff_latch_rec libc_ff_latch_rec;

/* ===== */

struct libc_internal_power /*COPY*/ {
    struct libc_name_list_rec *inputs;             /* old method */
    struct libc_name_list_rec *outputs;            /* old method */
    struct libc_name_list_rec *equal_or_opposite_output;
    text_buffer
    struct libc_name_list_rec *power_level;
    struct libc_name_list_rec *related_pin;
    struct libc_bool_opr_rec *when;

    struct libc_table_val_rec *rise_power;
    struct libc_table_val_rec *fall_power;
    struct libc_table_val_rec *power;                /* old method for
values */
    struct libc_internal_power *next;
}
```

libc_libcds.h

```
};

typedef struct libc_internal_power libc_internal_power;

/* ----- */

struct libc_leakage_power /*COPY*/ {
    struct libc_bool_opr_rec *when;
    float value;

    struct libc_leakage_power *next;
};

typedef struct libc_leakage_power libc_leakage_power;

/* ===== */

struct libc_memory_rec {
    /* text_buffer           *mem_name; */
    int is_ram;
    int address_width;
    int word_width;
};

typedef struct libc_memory_rec libc_memory_rec;

/* ===== */
/* ===== */

enum libc_timing_type_E {
    COMBINATIONAL_T,
    RISING_EDGE_T, FALLING_EDGE_T, PRESET_T, CLEAR_T, HOLD_RISING_T,
    HOLD_FALLING_T,
    SETUP_RISING_T, SETUP_FALLING_T, RECOVERY_RISING_T, RECOVERY_FALLING_T,
    THREE_STATE_DISABLE_T, THREE_STATE_ENABLE_T, REMOVAL_RISING_T,
    REMOVAL_FALLING_T,
    SKEW_RISING_T, SKEW_FALLING_T,
    NON_SEQ_HOLD_RISING_T, NON_SEQ_HOLD_FALLING_T,
    NON_SEQ_SETUP_RISING_T, NON_SEQ_SETUP_FALLING_T,
    NOCHANGE_HIGH_HIGH_T, NOCHANGE_HIGH_LOW_T, NOCHANGE_LOW_HIGH_T,
    NOCHANGE_LOW_LOW_T };
typedef enum libc_timing_type_E libc_timing_type_E;

enum libc_timing_sense_E { NON_UNATE_E, POSITIVE_UNATE_E, NEGATIVE_UNATE_E };
typedef enum libc_timing_sense_E libc_timing_sense_E;

/* ----- */

/* ---- for piecewise liner or ECL model only */
struct libc_piece_value_rec /*COPY*/ {
    int piece;
    float fall_delay_intercept;
    float fall_nonpaired_twin;
    float fall_pin_resistance;
    float fall_wire_resistance;
    float rise_delay_intercept;
    float rise_nonpaired_twin;
    float rise_pin_resistance;
    float rise_wire_resistance;
    struct libc_piece_value_rec *next;
}
```

libc_libcds.h

```
};

typedef struct libc_piece_value_rec libc_piece_value_rec;

/* ----- */

struct libc_float_rec /*COPY*/ {
    float          fv;
};

typedef struct libc_float_rec libc_float_rec;

/* ----- */

/* ---- for table lookup model only */
/* ---- 1D : values[0][0][i]
 *      2D : values[0][i][j]
 *      3D : values[i][j][k] */
struct libc_table_val_rec /*COPY*/ {
    struct libc_lu_table_template_rec *_tbl;           /* search it by name */
    float_buffer            *index1;
    float_buffer            *index2;
    float_buffer            *index3;
    float                  scalar_val;        /* for scalar table */
    float_buffer            *values;
};

typedef struct libc_table_val_rec libc_table_val_rec;

/* ----- */

struct libc_timing_rec /*COPY*/ {
    text_buffer            *t_name;
    struct libc_pin_rec   *_current_pin;

    /* ---- simple attr */
    float                 edge_rate_sensitivity_f0;
    float                 edge_rate_sensitivity_f1;
    float                 edge_rate_sensitivity_r0;
    float                 edge_rate_sensitivity_r1;
    float                 fall_resistance;
    float                 rise_resistance;
    float                 intrinsic_fall;
    float                 intrinsic_rise;
    struct libc_bool_opr_rec *related_bus_pins;
    enum variable_E       related_output_pin;
    struct libc_name_list_rec *related_pin;
    /* text_buffer          *sdf_cond; */
    /* text_buffer          *sdf_edge; */
    float                 slope_fall;
    float                 slope_rise;
    enum libc_timing_type_E timing_type;
    enum libc_timing_sense_E timing_sense;
    struct libc_bool_opr_rec *when;
    struct libc_bool_opr_rec *when_start;
    /* text_buffer          sdf_cond_start; */
    struct libc_bool_opr_rec *when_end;
    /* text_buffer          sdf_cond_end; */

    /* ---- complex attr */
}
```

libc_libcds.h

```
struct libc_piece_value_rec *piecewise;

/* ---- group stmt */
struct libc_table_val_rec    *cell_rise;
struct libc_table_val_rec    *cell_fall;
struct libc_table_val_rec    *rise_propagation;
struct libc_table_val_rec    *fall_propagation;
struct libc_table_val_rec    *rise_transition;
struct libc_table_val_rec    *fall_transition;
struct libc_table_val_rec    *rise_constraint;
struct libc_table_val_rec    *fall_constraint;

    struct libc_timing_rec      *next;
};

typedef struct libc_timing_rec libc_timing_rec;

/* ===== */

struct libc_min_pulse_width_rec /*COPY*/ {
    /* ---- simple attr */
    float                      constraint_high;
    float                      constraint_low;
    struct libc_bool_opr_rec   *when;
    /* text_buffer      sdf_cond; */
    struct libc_min_pulse_width_rec *next;
};
typedef struct libc_min_pulse_width_rec libc_min_pulse_width_rec;

/* ===== */

struct libc_minimum_period_rec /*COPY*/ {
    /* ---- simple attr */
    float                      constraint;
    struct libc_bool_opr_rec   *when;
    /* text_buffer      sdf_cond; */
    struct libc_minimum_period_rec *next;
};
typedef struct libc_minimum_period_rec libc_minimum_period_rec;

/* ===== */

struct libc_routing_track_rec {
    text_buffer                *layer_name;
    int                        tracks;
    float                      total_track_area;
    struct libc_routing_track_rec *next;
};
typedef struct libc_routing_track_rec libc_routing_track_rec;

/* ===== */

/* ----- */
/* ===== */
/* ===== */
/* ===== */

/* ---- using hash table (to keep the value) */
```

libc_libcds.h

```
enum value_type { REAL_VT, TEXT_VT, BOOL_VT };

struct libc_def_entry_rec {
    text_buffer           *def_name;
    enum value_type      v_type;
    struct libc_def_entry_rec *next;
};

typedef struct libc_def_entry_rec libc_def_entry_rec;

/* ----- */

struct libc_define_value_rec /*COPY*/ {                                /* for the attr value
*/
    struct libc_def_entry_rec *_def;
    float                  float_val;
    text_buffer            *str_val;
    struct libc_define_value_rec *next;
};

typedef struct libc_define_value_rec libc_define_value_rec;

/* ----- */

struct libc_glb_const_rec {
    text_buffer           *gc_name;
    float                 value;
    struct libc_glb_const_rec *next;
};

typedef struct libc_glb_const_rec libc_glb_const_rec;

/* ----- */

struct libc_def_table_rec {
    struct libc_glb_const_rec **gc_entry;
    struct libc_def_entry_rec  **lib;
    struct libc_def_entry_rec  **cell;
    struct libc_def_entry_rec  **pin;
};

typedef struct libc_def_table_rec libc_def_table_rec;

/* ====== */
```

702805 v1



APPENDIX B

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

bbdd.c

```
/*-----*/  
/*-----*/  
/*      File:          bbdd.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
#include <stdio.h>  
  
#include "blist.h"  
#include "bbdd.h"  
  
#ifndef STAT  
#define BDD_CALLOC      calloc  
#endif  
  
/*-----*/  
/* GLOBAL VARIABLE */  
/*-----*/  
BDD_WS      GLOB_BDD_WS;  
  
BDD_STATUS bdd_or ();  
BDD_STATUS bdd_impl ();  
BDD_STATUS bdd_and ();  
BDD_STATUS bdd_equiv ();  
  
APPLY_CACHE OrCache;  
APPLY_CACHE ImpCache;  
APPLY_CACHE AndCache;  
APPLY_CACHE EqCache;  
APPLY_CACHE RstCache;  
  
BDD_STATUS bdd_nand ();  
BDD_STATUS bdd_restrict ();  
BDD_STATUS bdd_swap_node ();  
  
/*-----*/  
/* CompareIndex */  
/*-----*/  
int CompareIndex (index, VarIndex)  
    Uint32      index;  
    Uint32      VarIndex;  
{  
  
    if (index == VarIndex)  
        return (0);  
  
    if (index < VarIndex)  
        return (-1);
```

bbdd.c

```
else
    return (1);
}

/*-----
/* CompareIndex_ec
/*-----*/
int CompareIndex_ec (index, VarIndex)
    Uint32          index;
    Uint32          VarIndex;
{
    if (index == VarIndex)
        return (0);

    /* The order is decreasing index variable
    if (index > VarIndex)
        return (-1);
    else
        return (1);
}

/*-----
/* ComplementBDD
/*-----*/
/* We complement only the pointer only if it not the BDD_Z and BDD_X */
/*-----*/
BDD ComplementBDD(x)
    BDD x;
{
    if ((x == BDD_X) || (x == BDD_Z))
        return (BDD_X);

    return (x ^ BDDTypeMask);
}

/*-----
/* GetBddPtrFromBdd
/*-----*/
BDD_PTR GetBddPtrFromBdd (Bbdd)
    BDD      Bbdd;
{
    return (GetBddPtr(Bbdd));
}

/*-----
/* GetBddPtrRefCount
/*-----*/
Uint32 GetBddPtrRefCount (BddPtr)
    BDD_PTR  BddPtr;
{
    return (((BddPtr)->Header) & RefCountMask);
}

/*-----
/* Access KEY functions for Caches (Hash Tables)
/*-----*/
```

bbdd.c

```
/*-----*/
#define APPLY_KEY_FUNCTION(v1,v2,size)  ( ( ((UInt32)v1 + (UInt32)v2)>>2 ) % size)

#define BDD_KEY_FUNCTION(b1,b2,size)    ( ( ((UInt32)b1>>4) + ((UInt32)b2>>2) ) % size)

/*-----*/
/* BddPtrIsVar */
/*-----*/
/* This function returns 1 if the BDD_PTR corresponds to a primaty IO */
/* otherwise returns 0. */
/*-----*/
int BddPtrIsVar (BddPtr)
    BDD_PTR BddPtr;
{

    if ((GetBddPtrEdge1 (BddPtr) == BDD_1) &&
        (GetBddPtrEdge0 (BddPtr) == BDD_0))
        return (1);

    return (0);
}

/*-----*/
/* bdd_one */
/*-----*/
BDD bdd_one ()
{
    return (BDD_1);
}

/*-----*/
/* bdd_zero */
/*-----*/
BDD bdd_zero ()
{
    return (BDD_0);
}

/*-----*/
/* bdd_z */
/*-----*/
BDD bdd_z ()
{
    return (BDD_Z);
}

/*-----*/
/* bdd_X */
/*-----*/
```

bbdd.c

```
BDD bdd_X ()
{
    return (BDD_X);
}

/*-----
/* ResetBddCache
*/
/*-----*/
/* This function removes all the BDD_PTR in the Bdd cache but keep safe */
/* all the buckets. Moreover all the Bdd corresponding to primary vars */
/* are preserved and stay in their bucket. */
*/
void ResetBddCache ()
{
    int             Index;
    int             i;
    BDD_PTR Bucket;
    BDD_PTR PreviousBucket;
    BLIST Cache;
    int             NbVar;
    BDD_PTR NextBucket;

    Cache = GLOB_BDD_WS->BddCache;
    NbVar = BListSize (Cache);

    for (Index = 0; Index < NbVar; Index++)
    {
        /* If at least one node in this bucket we need to scan it. */
        if (GetCache(Cache, Index)->NbBddNode)
        {
            for (i=0; i<GetCache (Cache, Index)->CacheSize; i++)
            {
                Bucket = GetCache (Cache, Index)->Buckets[i];
                PreviousBucket = NULL;
                while (Bucket != NULL)
                {
                    NextBucket = Bucket->Next;

                    /* we do not remove primary vars ! */
                    if (!BddPtrIsVar (Bucket))
                    {
                        if (PreviousBucket)
                        {
                            PreviousBucket->Next = NextBucket;
                        }
                        else
                        {
                            GetCache (Cache, Index)->Buckets[i] =
NextBucket;
                        }
                    }

                    /* We add the Bdd in the freed Bdd list */
                    (void)FreePhysicalBdd (Bucket);
                    (GetCache (Cache, Index)->NbBddNode)--;
                }
            }
        }
    }
}
```

```

        {
            PreviousBucket = Bucket;
        }
        Bucket = NextBucket;
    }
}

GetCache (Cache, Index) ->NbConflicts = 0;
GetCache (Cache, Index) ->NbDeadBddNode = 0;
}

GLOB_BDD_WS->TotalBddConflicts = 0;
GLOB_BDD_WS->TotalBddDead = 0;
GLOB_BDD_WS->TotalUsedBddNode = 0;
GLOB_BDD_WS->TotalCountBddNode = 0;

}

/*-----*/
/* SquizBddCache */
/*-----*/
/* This function removes all the BDD_PTR in the Bdd cache but keep safe */
/* all the buckets. Moreover all the Bdd corresponding to primary frozen */
/* Bdd nodes are preserved and stay in their bucket. */
/*-----*/
void SquizBddCache ()
{
    int             Index;
    int             i;
    BDD_PTR         Bucket;
    BDD_PTR         PreviousBucket;
    BLIST           Cache;
    int             NbVar;
    BDD_PTR         NextBucket;

    Cache = GLOB_BDD_WS->BddCache;
    NbVar = BListSize (Cache);

    for (Index = 0; Index < NbVar; Index++)
    {
        /* If at least one node in this bucket we need to scan it. */
        if (GetCache (Cache, Index) ->NbBddNode)
        {
            for (i=0; i<GetCache (Cache, Index) ->CacheSize; i++)
            {
                Bucket = GetCache (Cache, Index) ->Buckets[i];
                PreviousBucket = NULL;
                while (Bucket != NULL)
                {
                    NextBucket = Bucket->Next;

                    /* we do not remove Frozen bdd nodes ! */
                    if (!FrozenBddNode (Bucket))
                    {
                        if (PreviousBucket)

```

bbdd.c

```
        {
            PreviousBucket->Next = NextBucket;
        }
    else
    {
        GetCache (Cache, Index)->Buckets[i] =
NextBucket;
    }

    /* We add the Bdd in the freed Bdd list */
    (void)FreePhysicalBdd (Bucket);
    (GetCache (Cache, Index)->NbBddNode)--;
    (GLOB_BDD_WS->TotalUsedBddNode)--;
}
else
{
    PreviousBucket = Bucket;
}
Bucket = NextBucket;
}

}

GetCache (Cache, Index)->NbConflicts = 0;
GetCache (Cache, Index)->NbDeadBddNode = 0;
}

GLOB_BDD_WS->TotalBddConflicts = 0;
GLOB_BDD_WS->TotalBddDead = 0;

}

/*-----
/* bdd_swap_buckets
/*-----
/* never tested !
/*-----*/
BDD_STATUS bdd_swap_buckets (BddCache, Index)
    BDD_CACHE      BddCache;
    UInt32        Index;
{
    int           i;
    BDD_PTR      Bucket;
    int           Gain;
    BDD           BddRes;
    BDD_STATUS    Status;

    for (i=0; i<BddCache[Index].CacheSize; i++)
    {
        Bucket = BddCache[Index].Buckets[i];
        while (Bucket != NULL)
        {
            if ((Status = bdd_swap_node (Bucket, Index, Index+1,
                                         Gain, &BddRes)))
                return (Status);
        }
    }
}
```

bbdd.c

```
        Bucket = Bucket->Next;
    }
}

return (BDD_OK);
}

/*-----
/* bdd_swap
*/
/* This functions performs the swapping in the Bdd workspace "Ws"
 * between Vars of index "i" and "i+1".
*/
BDD_STATUS bdd_swap (Ws, i)
    BDD_WS    Ws;
    Uint32   i;
{
    BDD_STATUS      Status;
    BLIST     Cache;

/*
    if (i >= Ws->MaxNbVar-1)
        return (BDD_OK);

    Cache = GLOB_BDD_WS->BddCache;

    if ((Status = bdd_swap_buckets (Cache, i)))
        return (Status);

    return (BDD_OK);
*/
}

/*-----
/* ResetApplyCache
*/
/* We reset the Apply cache by removing all the APPLY CELL objects and
 * freeing the Bdds on which they point.
*/
static void ResetApplyCache (Cache)
    APPLY_CACHE    Cache;
{
    int           i;
    APPLY_CELL    Bucket;
    APPLY_CELL    NextBucket;

    for (i=0; i<Cache->CacheSize; i++)
    {
        Bucket = Cache->Buckets[i];
        while (Bucket != NULL)
        {
            NextBucket = Bucket->Next;
            FreeBdd (Bucket->Bdd1); /* We free the pointed BDD */
            FreeBdd (Bucket->Bdd2); /* We free the pointed BDD */
```

```

        FreeBdd (Bucket->BddRes); /* We free the pointed BDD      */
        FreeApplyCell (Bucket);
        Bucket = NextBucket;
    }
    Cache->Buckets[i] = NULL;
}

Cache->NbConflicts = 0;
}

/*-----
/* QuickResetApplyCache                               */
/*-----*/
/* We reset the Apply cache by removing all the APPLY CELL objects. We   */
/* not care to Free the BDD on which they point, because we want a quick*/
/* reset (because we suppose that "ResetBddCache" is also called).       */
/*-----*/
static void QuickResetApplyCache (Cache)
    APPLY_CACHE    Cache;
{
    int             i;
    APPLY_CELL     Bucket;
    APPLY_CELL     NextBucket;

    for (i=0; i<Cache->CacheSize; i++)
    {
        Bucket = Cache->Buckets[i];
        while (Bucket != NULL)
        {
            NextBucket = Bucket->Next;
            FreeApplyCell (Bucket);
            Bucket = NextBucket;
        }
        Cache->Buckets[i] = NULL;
    }

    Cache->NbConflicts = 0;
}

/*-----
/* CompactApplyCache                                */
/*-----*/
/* We remove APPLY CELL where the BddRes field has a ref count of 1.    */
/* This means that this BddRes is used only by the APPLY CELL and is not*/
/* usefull anymore. So we can remove the corresponding APPLY CELL and    */
/* free the pointed BDDs.                                              */
/*-----*/
static void CompactApplyCache (Cache)
    APPLY_CACHE    Cache;
{
    int             i;
    APPLY_CELL     Bucket;
    APPLY_CELL     NextBucket;
    APPLY_CELL     PrevBucket;

```

```

PrevBucket = NULL;
for (i=0; i<Cache->CacheSize; i++)
{
    Bucket = Cache->Buckets[i];
    while (Bucket != NULL)
    {
        if (GetBddPtrRefCount (
            GetBddPtrFromBdd (Bucket->BddRes)) == 1)
        {
            NextBucket = Bucket->Next;
            FreeBdd (Bucket->Bdd1); /* We free the pointed BDD */
            FreeBdd (Bucket->Bdd2); /* We free the pointed BDD */
            FreeBdd (Bucket->BddRes);/* We free the pointed BDD */
            FreeApplyCell (Bucket);

            /* case we removed the very first element */
            if (PrevBucket == NULL)
            {
                Cache->Buckets[i] = NextBucket;
                Bucket = NextBucket;
            }
            else
            {
                PrevBucket = Bucket;
                Bucket = NextBucket;
            }
        }
        else
        {
            PrevBucket = Bucket;
            Bucket = Bucket->Next;
        }
    }
}

/*
 * ResetAllApplyCaches
 */
void ResetAllApplyCaches ()
{
    (void)ResetApplyCache (GLOB_BDD_WS->OrCache);
    (void)ResetApplyCache (GLOB_BDD_WS->AndCache);
    (void)ResetApplyCache (GLOB_BDD_WS->ImpCache);
    (void)ResetApplyCache (GLOB_BDD_WS->EqCache);
    (void)ResetApplyCache (GLOB_BDD_WS->RstCache);
    (void)ResetApplyCache (GLOB_BDD_WS->JoinCache);

    GLOB_BDD_WS->TotalApplyConflicts = 0;
    GLOB_BDD_WS->TotalApplySuccess = 0;
}

```

bbdd.c

```
/*-----*/
/* QuickResetAllApplyCaches */ */
/*-----*/
void QuickResetAllApplyCaches ()
{
    (void)QuickResetApplyCache (GLOB_BDD_WS->OrCache);
    (void)QuickResetApplyCache (GLOB_BDD_WS->AndCache);
    (void)QuickResetApplyCache (GLOB_BDD_WS->ImpCache);
    (void)QuickResetApplyCache (GLOB_BDD_WS->EqCache);
    (void)QuickResetApplyCache (GLOB_BDD_WS->RstCache);
    (void)QuickResetApplyCache (GLOB_BDD_WS->JoinCache);

    GLOB_BDD_WS->TotalApplyConflicts = 0;
    GLOB_BDD_WS->TotalApplySuccess = 0;

}

/*-----*/
/* CompactAllApplyCaches */ */
/*-----*/
void CompactAllApplyCaches ()
{
    (void)CompactApplyCache (GLOB_BDD_WS->OrCache);
    (void)CompactApplyCache (GLOB_BDD_WS->AndCache);
    (void)CompactApplyCache (GLOB_BDD_WS->ImpCache);
    (void)CompactApplyCache (GLOB_BDD_WS->EqCache);
    (void)CompactApplyCache (GLOB_BDD_WS->RstCache);
    (void)CompactApplyCache (GLOB_BDD_WS->JoinCache);
}

/*-----*/
/* ResetWorkSpace */ */
/*-----*/
void ResetWorkSpace (Ws)
    BDD_WS    Ws;
{
    GLOB_BDD_WS = Ws;
    QuickResetAllApplyCaches ();
    ResetBddCache ();
}

/*-----*/
/* ResetDefaultWorkSpace */ */
/*-----*/
void ResetDefaultWorkSpace ()
{
    QuickResetAllApplyCaches ();
    ResetBddCache ();
}

/*-----*/
/* SquizWorkSpace */ */
/*-----*/
```

bbdd.c

```
/*-----*/
/* This function removes from the Bdd work space all the Bdd nodes which*/
/* are not frozen. It removes also completely the Apply cache tables.      */
/*-----*/
void SquizWorkSpace (Ws)
    BDD_WS    Ws;
{

    GLOB_BDD_WS = Ws;
    QuickResetAllApplyCaches ();
    SquizBddCache ();
}

/*-----*/
/* SquizDefaultWorkSpace                               */
/*-----*/
/* This function removes from the Bdd work space all the Bdd nodes which*/
/* are not frozen. It removes also completely the Apply cache tables.      */
/*-----*/
void SquizDefaultWorkSpace ()
{
    QuickResetAllApplyCaches ();
    SquizBddCache ();
}

/*-----*/
/* ReCreateApplyCache                                */
/*-----*/
/* This function recreates a new APPLY_CACHE with a new size .           */
/*-----*/
BDD_STATUS ReCreateApplyCache (Cache)
    APPLY_CACHE    Cache;
{
    int             i;
    int             PreviousSize;
    APPLY_CELL     *NewBuckets;
    APPLY_CELL     Bucket;
    APPLY_CELL     NextBucket;
    Uint32          Key;

    if (Cache->CacheSize > MAX_APPLY_CACHE_SIZE)
    {
        ResetApplyCache (Cache);
        Cache->NbConflicts = 0;
        return (BDD_OK);
    }

    if ((NewBuckets = (APPLY_CELL*) BDD_CALLOC (4*Cache->CacheSize,
                                                sizeof(APPLY_CELL))) == NULL)
        return (BDD_MEMORY_FULL);

    PreviousSize = Cache->CacheSize;
    Cache->CacheSize = 4*Cache->CacheSize;

    for (i=0; i<PreviousSize; i++)
    {
```

bbdd.c

```
Bucket = Cache->Buckets[i];
while (Bucket != NULL)
{
    NextBucket = Bucket->Next;

    Key = APPLY_KEY_FUNCTION (Bucket->Bdd1, Bucket->Bdd2,
        Cache->CacheSize);

    Bucket->Next = NewBuckets[Key];
    NewBuckets[Key] = Bucket;

    Bucket = NextBucket;
}
}

free ((char*)Cache->Buckets);
GLOB_BDD_WS->TotalMemoryAlloc -= PreviousSize*sizeof(APPLY_CELL);

Cache->Buckets = NewBuckets;

Cache->NbConflicts = 0;

return (BDD_OK);
}

/*-----
 * GetCommuteApplyInCache
 *-----
 * Bdd1 and Bdd2 are arguments of a commutative operator so we try to
 * match both (Bdd1, Bdd2) and (Bdd2, Bdd1).
 *-----*/
static BDD GetCommuteApplyInCache (Cache, Bucket, Bdd1, Bdd2, NbCells)
    APPLY_CACHE    Cache;
    APPLY_CELL     Bucket;
    BDD_PTR        Bdd1;
    BDD_PTR        Bdd2;
    int            *NbCells;
{
    *NbCells = 0;

    for (; Bucket != NULL; Bucket = Bucket->Next)
    {
        (*NbCells)++;

        if ((Bucket->Bdd1 == Bdd1 && Bucket->Bdd2 == Bdd2) ||
            (Bucket->Bdd1 == Bdd2 && Bucket->Bdd2 == Bdd1))
            return (Bucket->BddRes);

        Cache->NbConflicts++;

#ifdef STAT
        GLOB_BDD_WS->TotalApplyConflicts++;
#endif
    }

    return (BDD_NULL);
```

```

}

/*-----*/
/* GetNoCommutApplyInCache */ */
/*-----*/
/* Bdd1 and Bdd2 are arguments of a non-commutative operator so we try */
/* to match only wth the couple (Bdd1, Bdd2). */ */
/*-----*/
static BDD GetNoCommutApplyInCache (Cache, Bucket, Bdd1, Bdd2,
NbCells)
{
    APPLY_CACHE Cache;
    APPLY_CELL Bucket;
    BDD_PTR Bdd1;
    BDD_PTR Bdd2;
    int *NbCells;
{
    *NbCells = 0;

    for (; Bucket != NULL; Bucket = Bucket->Next)
    {
        (*NbCells)++;
        if (Bucket->Bdd1 == Bdd1 && Bucket->Bdd2 == Bdd2)
            return (Bucket->BddRes);

        Cache->NbConflicts++;

#ifdef STAT
        GLOB_BDD_WS->TotalApplyConflicts++;
#endif
    }

    return (BDD_NULL);
}

/*-----*/
/* bdd_freeze */ */
/*-----*/
/* Set the reference counter of bdd node "bdd" to BDD_MAXREFCOUNT in */
/* order to freeze it. */ */
/*-----*/
void bdd_freeze (Bdd)
    BDD Bdd;
{
    SetBddPtrRefCount (GetBddPtrFromBdd (Bdd), BDD_MAXREFCOUNT);
}

/*-----*/
/* UseBddPtr */ */
/*-----*/
/* Increments the reference counter of the bdd ptr node "BddPtr". */
/*-----*/
/* CAUTION!: we can optimize all this !!! */ */
/*-----*/
BDD_PTR UseBddPtr (BddPtr)
    BDD_PTR BddPtr;
{
}

```

bbdd.c

```
register Uint32      counter;

counter = GetBddPtrRefCount (BddPtr);

if (counter != BDD_MAXREFCOUNT)
{
    counter++;
    SetBddPtrRefCount (BddPtr, counter);
}

return (BddPtr);
}

/*-----
/* UseBdd
/*-----
/* Increments the reference counter of the bdd node "Bdd".
/* CAUTION!: we can optimize all this !!!
/*-----*/
BDD UseBdd (Bdd)
    BDD Bdd;
{
    register BDD_PTR      BddPtr;

    BddPtr = UseBddPtr (GetBddPtrFromBdd (Bdd));
    return (Bdd);
}

/*-----
/* ReCreateBddCache
/*-----
BDD_STATUS ReCreateBddCache (Cache, Index)
    BLIST      Cache;
    int         Index;
{
    int          i;
    int          PreviousSize;
    BDD_PTR     *NewBuckets;
    BDD_PTR     Bucket;
    BDD_PTR     NextBucket;
    Uint32      Key;

    if (GetCache (Cache, Index) ->CacheSize > MAX_BDD_CACHE_SIZE)
        return (BDD_OK);

    if ((NewBuckets = (BDD_PTR*)BDD_CALLOC (
                    4*GetCache (Cache, Index) ->CacheSize,
                    sizeof(BDD_PTR))) == NULL)
        return (BDD_MEMORY_FULL);

    PreviousSize = GetCache (Cache, Index) ->CacheSize;
```

```

GetCache (Cache,Index) ->CacheSize = 4*GetCache (Cache,Index) ->CacheSize;

for (i=0; i<PreviousSize; i++)
{
    Bucket = GetCache (Cache,Index) ->Buckets[i];
    while (Bucket != NULL)
    {
        NextBucket = Bucket->Next;

        Key = BDD_KEY_FUNCTION (Bucket->Edge[1], Bucket->Edge[0],
                               GetCache (Cache,Index) ->CacheSize);
        Bucket->Next = NewBuckets[Key];
        NewBuckets[Key] = Bucket;

        Bucket = NextBucket;
    }
}

free ((char*)GetCache (Cache,Index) ->Buckets);
GLOB_BDD_WS->TotalMemoryAlloc -= PreviousSize*sizeof(BDD_PTR);

GetCache (Cache,Index) ->Buckets = NewBuckets;

GetCache (Cache,Index) ->NbConflicts = 0;

return (BDD_OK);
}

/*
 * AddOrGetBddInCache
 */
/*
 * PREC: ListBucket stores BDD_PTR with index "index".
 * "BddRes" is a BDD_PTR (e.g. BDD always positive).
 * The "BddRes" is a protected Bdd where we have incremented its ref.
 * counter.
 */
BDD_STATUS AddOrGetBddInCache (Cache, index, Edge1, Edge0, BddRes)
{
    BLIST Cache;
    Uint32 index;
    BDD Edge1;
    BDD Edge0;
    BDD_PTR *BddRes;
    BDD_STATUS Status;
    BDD_PTR NewBdd;
    BDD_PTR Bucket;
    Uint32 Key;

    Key = BDD_KEY_FUNCTION (Edge1, Edge0, GetCache (Cache,index) ->CacheSize);

    Bucket = (GetCache (Cache,index) ->Buckets)[Key];

    if (Bucket)
        while (1)
        {
            if (GetBddPtrEdge1 (Bucket) == Edge1 &&

```

```

        GetBddPtrEdge0 (Bucket) == Edge0)
    {
        *BddRes = UseBddPtr (Bucket);
#endif STAT
        GLOB_BDD_WS->TotalApplySuccess++;
#endif
        return (BDD_OK);
    }
    if (GetBddPtrNext (Bucket) == BDD_PTR_NULL)
        break;
    Bucket = GetBddPtrNext (Bucket);

    (GetCache (Cache, index)->NbConflicts)++;
#endif STAT
    GLOB_BDD_WS->TotalBddConflicts++;
#endif
}

/* Not found, so we create physically a new BDD_PTR */ *
if ((Status = CreateBddNode (&NewBdd)))
    return (Status);

/* Setting the fields ... */ /
SetBddPtrIndex (NewBdd, index);
SetBddPtrEdge1 (NewBdd, Edge1);
SetBddPtrEdge0 (NewBdd, Edge0);

if (Bucket)
    SetBddPtrNext (Bucket, NewBdd);
else
    (GetCache (Cache, index)->Buckets)[Key] = NewBdd;

(GetCache (Cache, index)->NbBddNode)++;

*BddRes = UseBddPtr (NewBdd);

if (GetCache (Cache, index)->NbConflicts >
    GetCache (Cache, index)->CacheSize << 2)
    ReCreateBddCache (Cache, index);

return (BDD_OK);
}

/*
 * MakePositiveBdd
 */
/* PREC: Edge1 != Edge0 */ /
/* "BddRes" is always positive (it is a BDD_PTR) */ /
/* The "BddRes" is protected and had its ref. counter incremented. */ /
/* */

BDD_STATUS MakePositiveBdd (index, Edge1, Edge0, BddRes)
    Uint32 index;
    BDD Edge1;
    BDD Edge0;
    BDD_PTR *BddRes;

```

bbdd.c

```
{  
    BDD_STATUS      Status;  
  
    if ((Status = AddOrGetBddInCache (GLOB_BDD_WS->BddCache,  
                                      index, Edge1, Edge0, BddRes)))  
        return (Status);  
  
    return (BDD_OK);  
}  
  
/*-----*/  
/* MakeBdd */  
/*-----*/  
/* The routine MakeBdd creates a new typed BDD from the index of its */  
/* root variable, its Edge1 branch and its Edge0 branch. The new BDD can */  
/* be positive or negative. */  
/* By convention to preserve canonicity, the Edge1 edge must always be */  
/* of type direct. */  
/*-----*/  
BDD_STATUS MakeBdd (index, Edge1, Edge0, BddRes)  
    Uint32      index;  
    BDD         Edge1;  
    BDD         Edge0;  
    BDD         *BddRes;  
{  
    BDD_STATUS      Status;  
  
    if (Edge1 == Edge0)  
    {  
        *BddRes = UseBdd (Edge0);  
        return(BDD_OK);  
    }  
  
    if (IsBddTypeINV (Edge1)) /* Edge1 son type must always be DIRECT */  
    {  
        BDD_PTR      Res;  
  
        /* The "Res" Bdd is protected and had its ref. counter incremented*/  
        if ((Status = MakePositiveBdd (index,  
                                       ComplementBDD (Edge1),  
                                       ComplementBDD (Edge0),  
                                       &Res)))  
            return (Status);  
  
        *BddRes = ComplementBDD (GetBddFromBddPtr (Res));  
        return(BDD_OK);  
    }  
  
    /* The "Res" Bdd is protected and had its ref. counter incremented. */  
    if ((Status = MakePositiveBdd (index, Edge1, Edge0, (BDD_PTR*)BddRes)))  
        return (Status);  
}
```

bbdd.c

```
return(BDD_OK);

}

/*-----*/
/* ApplyCommutOp */
/*-----*/
/* Performs logical operation on two positive BDDs. Operation is */
/* Commutative. */
/*-----*/
static BDD_STATUS ApplyCommutOp (fun, v1, v2, cache, BddRes)
    BDD_STATUS      (fun) ();
    BDD_PTR         v1;
    BDD_PTR         v2;
    APPLY_CACHE     cache;
    BDD             *BddRes;
{
    BDD             result;
    BDD_STATUS      Status;
    Uint32          Key;
    Uint32          NbCells;

    Key = APPLY_KEY_FUNCTION (v1, v2,
                              cache->CacheSize);

    result = GetCommutApplyInCache (cache, cache->Buckets [Key], v1, v2,
                                    &NbCells);

    if (result != BDD_NULL)
    {
        *BddRes = UseBdd (result);
        return (BDD_OK);
    }
    else
    {
        Uint32          index1 = GetBddPtrIndex (v1);
        Uint32          index2 = GetBddPtrIndex (v2);
        BDD_STATUS      Status;
        BDD             Bdd1;
        BDD             Bdd2;

        switch ((GLOB_BDD_WS->CompareIdx) (index1, index2))
        {
            case -1:
                if ((Status = (fun) (GetBddPtrEdge1 (v1), (BDD)v2,
                                     &Bdd1)))
                    return (Status);

                if ((Status = (fun) (GetBddPtrEdge0 (v1), (BDD)v2,
                                     &Bdd2)))
                    return (Status);

                if ((Status = MakeBdd (index1, Bdd1, Bdd2, BddRes)))
                    return (Status);
        }
    }
}
```

```

        break;

    case 0:
        if ((Status = (fun) (GetBddPtrEdge1 (v1),
                            GetBddPtrEdge1 (v2), &Bdd1)))
            return (Status);

        if ((Status = (fun) (GetBddPtrEdge0 (v1),
                            GetBddPtrEdge0 (v2), &Bdd2)))
            return (Status);

        if ((Status = MakeBdd (index1, Bdd1, Bdd2, BddRes)))
            return (Status);

        break;

    case 1:
        if ((Status = (fun) ((BDD)v1, GetBddPtrEdge1 (v2),
                            &Bdd1)))
            return (Status);

        if ((Status = (fun) ((BDD)v1, GetBddPtrEdge0 (v2),
                            &Bdd2)))
            return (Status);

        if ((Status = MakeBdd (index2, Bdd1, Bdd2, BddRes)))
            return (Status);

        break;
    }

    if (NbCells < MAX_NB_APPLY_CELLS)
    {
        APPLY_CELL      NewApplyCell;

        if ((Status = CreateApplyCell (&NewApplyCell)))
            return (Status);

        NewApplyCell->Bdd1 =  UseBddPtr (v1);
        NewApplyCell->Bdd2 =  UseBddPtr (v2);
        NewApplyCell->BddRes = UseBdd (*BddRes);
        NewApplyCell->Next = cache->Buckets[Key];
        cache->Buckets[Key] = NewApplyCell;
    }

    if (cache->NbConflicts > cache->CacheSize << 2)
        ReCreateCache (cache);

    return (BDD_OK);
}
/* ApplyCommOp */
/*
 * ApplyNoCommOp
 */

```

bbdd.c

```
/* Performs logical operation on two positive BDDs. Operation is */
/* not Commutative. */
```

```
static BDD_STATUS ApplyNoCommutOp (fun, v1, v2, cache, BddRes)
    BDD_STATUS      (fun) ();
    BDD_PTR         v1;
    BDD_PTR         v2;
    APPLY_CACHE     cache;
    BDD             *BddRes;
{
    BDD             result;
    int             Key;
    int             NbCells;

    Key = APPLY_KEY_FUNCTION (v1,v2,
                               cache->CacheSize);

    result = GetNoCommuteApplyInCache (cache, cache->Buckets[Key],
                                       v1, v2, &NbCells);

    if (result != BDD_NULL)
    {
        *BddRes = UseBdd (result);
        return (BDD_OK);
    }
    else
    {
        Uint32          index1 = GetBddPtrIndex (v1);
        Uint32          index2 = GetBddPtrIndex (v2);
        BDD_STATUS      Status;
        BDD             Bdd1;
        BDD             Bdd2;

        switch ((GLOB_BDD_WS->CompareIdx) (index1, index2))
        {
            case -1:
                if ((Status = (fun) (GetBddPtrEdge1 (v1),
                                      GetBddFromBddPtr (v2), &Bdd1)))
                    return (Status);

                if ((Status = (fun) (GetBddPtrEdge0 (v1),
                                      GetBddFromBddPtr (v2), &Bdd2)))
                    return (Status);

                if ((Status = MakeBdd (index1, Bdd1, Bdd2, BddRes)))
                    return (Status);

                break;

            case 0:
                if ((Status = (fun) (GetBddPtrEdge1 (v1),
                                      GetBddPtrEdge1 (v2), &Bdd1)))
                    return (Status);
```

```

        if ((Status = (fun) (GetBddPtrEdge0 (v1),
                             GetBddPtrEdge0 (v2), &Bdd2)))
            return (Status);

        if ((Status = MakeBdd (index1, Bdd1, Bdd2, BddRes)))
            return (Status);

        break;

    case 1:
        if ((Status = (fun) (GetBddFromBddPtr (v1),
                             GetBddPtrEdge1 (v2), &Bdd1)))
            return (Status);

        if ((Status = (fun) (GetBddFromBddPtr (v1),
                             GetBddPtrEdge0 (v2), &Bdd2)))
            return (Status);

        if ((Status = MakeBdd (index2, Bdd1, Bdd2, BddRes)))
            return (Status);

        break;
    }

    if (NbCells < MAX_NB_APPLY_CELLS)
    {
        APPLY_CELL      NewApplyCell;

        if ((Status = CreateApplyCell (&NewApplyCell)))
            return (Status);

        NewApplyCell->Bdd1 = UseBddPtr(v1);
        NewApplyCell->Bdd2 = UseBddPtr(v2);
        NewApplyCell->BddRes = UseBdd (*BddRes);
        NewApplyCell->Next = cache->Buckets[Key];
        cache->Buckets[Key] = NewApplyCell;
    }

    if (cache->NbConflicts > cache->CacheSize << 2)
        ReCreateApplyCache (cache);

    return (BDD_OK);
}
}                                /* ApplyNocommOp */

/*-----*/
/* ApplyRestrict                      */
/*-----*/
static BDD_STATUS ApplyRestrict (vtx, dmn, cache, BddRes)
    BDD_PTR  vtx;
    BDD       dmn;
    APPLY_CACHE  cache;
    BDD       *BddRes;
{
    BDD      result;
    int      Key;

```

```

int      NbCells;
APPLY_CELL      NewApplyCell;

Key = APPLY_KEY_FUNCTION (vtx, GetBddPtrFromBdd (dmn),
                         cache->CacheSize);

result = GetNoCommitApplyInCache (cache, cache->Buckets [Key],
                                  vtx, GetBddPtrFromBdd (dmn), &NbCells);

if (result != BDD_NULL)
{
    *BddRes = UseBdd (result);
    return (BDD_OK);
}
else
{
    BDD_PTR          vtxd      = GetBddPtrFromBdd (dmn);
    Uint32           index     = GetBddPtrIndex (vtx);
    Uint32           indexd    = GetBddPtrIndex (vtxd);
    BDD             Bdd1;
    BDD             Bdd2;
    BDD_STATUS       Status;

    switch ((GLOB_BDD_WS->CompareIdx) (index, indexd))
    {
        case -1:
            if ((Status = bdd_restrict (GetBddPtrEdge1 (vtx), dmn,
                                         &Bdd1)))
                return (Status);

            if ((Status = bdd_restrict (GetBddPtrEdge0 (vtx), dmn,
                                         &Bdd2)))
                return (Status);

            if ((Status = MakeBdd (index, Bdd1, Bdd2, BddRes)))
                return (Status);

            break;

        case 0:
            if (IsBddTypeINV(dmn))
            {
                if (GetBddPtrEdge1 (vtxd) == BDD_1)
                {
                    if ((Status = bdd_restrict (GetBddPtrEdge0 (vtx),
                                              ComplementBDD (GetBddPtrEdge0
(vtxd)),
                                              BddRes)))
                        return (Status);
                }
                else if (GetBddPtrEdge0 (vtxd) == BDD_1)
                {
                    if ((Status = bdd_restrict (GetBddPtrEdge1 (vtx),
                                              ComplementBDD (GetBddPtrEdge1
(vtxd)),
                                              BddRes)))
                        return (Status);
                }
            }
    }
}

```

```

                                BddRes)))
    return (Status);
}
else
{
    if ((Status = bdd_restrict (GetBddPtrEdge1 (vtx),
                                ComplementBDD (GetBddPtrEdge1
(vtxd)),
                                &Bdd1)))
        return (Status);
    if ((Status = bdd_restrict (GetBddPtrEdge0 (vtx),
                                ComplementBDD (GetBddPtrEdge0
(vtxd)),
                                &Bdd2)))
        return (Status);

    if ((Status = bdd_or (Bdd1, Bdd2, BddRes)))
        return (Status);
}
else
{
    if (GetBddPtrEdge1 (vtxd) == BDD_0)
    {
        if ((Status = bdd_restrict (GetBddPtrEdge0 (vtx),
                                    GetBddPtrEdge0 (vtxd),
                                    BddRes)))
            return (Status);
    }
    else
    {
        if ((Status = bdd_restrict (GetBddPtrEdge1 (vtx),
                                    GetBddPtrEdge1 (vtxd),
                                    &Bdd1)))
            return (Status);
        if ((Status = bdd_restrict (GetBddPtrEdge0 (vtx),
                                    GetBddPtrEdge0 (vtxd),
                                    &Bdd2)))
            return (Status);

        if ((Status = bdd_or (Bdd1, Bdd2, BddRes)))
            return (Status);
    }
}
break;

case 1:
    if (IsBddTypeINV (dmn))
    {
        if (GetBddPtrEdge1 (vtxd) == BDD_1)
        {
            if ((Status = bdd_restrict ((BDD)vtx,
                            ComplementBDD (GetBddPtrEdge0
(vtxd)),
                            BddRes)))
                return (Status);
        }
    }
}

```

```

        }
    else if (GetBddPtrEdge0 (vtxd) == BDD_1)
    {
        if ((Status = bdd_restrict ((BDD)vtx,
                                    ComplementBDD (GetBddPtrEdge1 (vtxd)),
                                    BddRes)))
            return (Status);
    }
else
{
    BDD  NewDmn;

    if ((Status = bdd_nand (GetBddPtrEdge1 (vtxd),
                            ComplementBDD (GetBddPtrEdge0
(vtxd)),
                            &NewDmn)))
        return (Status);

    if ((Status = bdd_restrict ((BDD)vtx,
                               NewDmn,
                               BddRes)))
        return (Status);

    FreeBdd (NewDmn);
}
else
{
    if (GetBddPtrEdge1 (vtxd) == BDD_0)
    {
        if ((Status = bdd_restrict ((BDD)vtx,
                                    GetBddPtrEdge0 (vtxd),
                                    BddRes)))
            return (Status);
    }
else
{
    BDD  NewDmn;

    if ((Status = bdd_or (GetBddPtrEdge1 (vtxd),
                          GetBddPtrEdge0 (vtxd),
                          &NewDmn)))
        return (Status);

    if ((Status = bdd_restrict ((BDD)vtx,
                               NewDmn,
                               BddRes)))
        return (Status);

    FreeBdd (NewDmn);
}
}

break;
}

if (NbCells < MAX_NB_APPLY_CELLS)

```

```

    {
        if ((Status = CreateApplyCell (&NewApplyCell)))
            return (Status);

        NewApplyCell->Bdd1 = UseBddPtr (vtx);
        /* may be a PROBLEM here ... */
        NewApplyCell->Bdd2 = GetBddPtrFromBdd (UseBdd (dmn));
        NewApplyCell->BddRes = UseBdd (*BddRes);
        NewApplyCell->Next = cache->Buckets [Key];
        cache->Buckets [Key] = NewApplyCell;
    }

    if (cache->NbConflicts > cache->CacheSize << 2)
        ReCreateApplyCache (cache);

    return (BDD_OK);
}
}                                /* ApplyRestrict */

/*-----*
/* bdd_join
/*-----*
/*
/* -- resolution function
/*-----*
/* CONSTANT resolution_table : stdlogic_table := (
/* -- -----
/* -- | U X 0 1 Z W L H - | | |
/* -- -----
/* -- ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
/* -- ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
/* -- ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
/* -- ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
/* -- ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
/* -- ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
/* -- ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
/* -- ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
/* -- ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
/*-----*
/*
/* For 4 values (1, 0, Z, X) we extract:
/*-----*
/* | 1 0 Z X | | |
/*-----*
/* ( '1', 'X', '1', 'X' ) | 1 | */
/* ( 'X', '0', '0', 'X' ) | 0 | */
/* ( '1', '0', 'Z', 'X' ) | Z | */
/* ( 'X', 'X', 'X', 'X' ) | X | */
/*-----*
BDD_STATUS bdd_join (Bdd1, Bdd2, BddRes)
    BDD      Bdd1;
    BDD      Bdd2;
    BDD      *BddRes;
{
    BDD      Res;
}

```

```

BDD_STATUS      Status;

if ((Bdd1 == BDD_X) || (Bdd2 == BDD_X))
{
    *BddRes = BDD_X;
    return (BDD_OK);
}

if (Bdd1 == BDD_Z)
{
    *BddRes = UseBdd (Bdd2);
    return (BDD_OK);
}

if (Bdd2 == BDD_Z)
{
    *BddRes = UseBdd (Bdd1);
    return (BDD_OK);
}

if ((Bdd1 == BDD_1) && (Bdd2 == BDD_0))
{
    *BddRes = BDD_X;
    return (BDD_OK);
}

if ((Bdd1 == BDD_0) && (Bdd2 == BDD_1))
{
    *BddRes = BDD_X;
    return (BDD_OK);
}

if (IsBddTypeINV (Bdd1))
{
    if (IsBddTypeINV (Bdd2))
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_join,
                                      GetBddPtrFromBdd (Bdd1),
                                      GetBddPtrFromBdd (Bdd2),
                                      GLOB_BDD_WS->OrCache,
                                      BddRes)))
            return (Status);

        return (BDD_OK);
    }
    else
    {
        if (GetBddPtrFromBdd (Bdd1) == (BDD_PTR) Bdd2)
        {
            *BddRes = BDD_0;
        }
    }
}

```

```

        return (BDD_OK);
    }
    if ((Status = ApplyCommutOp (bdd_join,
                                 (BDD_PTR)Bdd2,
                                 GetBddPtrFromBdd (Bdd1),
                                 GLOB_BDD_WS->ImpCache,
                                 &Res)))
        return (Status);

    if (Res == BDD_Z)
        *BddRes = BDD_Z;
    else
        *BddRes = ComplementBDD(Res);
    return (BDD_OK);
}
else
{
    if (IsBddTypeINV (Bdd2))
    {
        if ((BDD_PTR)Bdd1 == GetBddPtr(Bdd2))
        {
            *BddRes = BDD_X;           /* Bdd1 and Bdd2 are complement
*/            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_join,
                                     (BDD_PTR)Bdd1,
                                     GetBddPtr (Bdd2),
                                     GLOB_BDD_WS->ImpCache,
                                     &Res)))
            return (Status);

        if (Res == BDD_Z)
            *BddRes = BDD_Z;
        else
            *BddRes = ComplementBDD(Res);
        return (BDD_OK);
    }
    else
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_join,
                                     (BDD_PTR)Bdd1,
                                     (BDD_PTR)Bdd2,
                                     GLOB_BDD_WS->JoinCache,
                                     BddRes)))
            return (Status);

        return (BDD_OK);
    }
}

```

```

    }

}

/*-----*/
/* bdd_and */
/*-----*/
/* Logical Operator AND following the std_logic_1164 table. */
/* -- truth table for "and" function */
/* CONSTANT and_table : stdlogic_table := (
/*                                     */
/* --   ----- */
/* -- | U  X  0  1  Z  W  L  H  -  |  | */
/* -- ----- */
/* -- ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U */
/* -- ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X */
/* -- ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 */
/* -- ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 */
/* -- ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z */
/* -- ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W */
/* -- ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L */
/* -- ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H */
/* -- ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ) -- | - */
/*                                     */
/* For 4 values (1, 0, Z, X) we extract: */
/*   */
/*   | 1  0  Z  X  |  | */
/*   ----- */
/*   ( '1', '0', 'X', 'X' ) | 1 */
/*   ( '0', '0', '0', '0' ) | 0 */
/*   ( 'X', '0', 'X', 'X' ) | Z */
/*   ( 'X', '0', 'X', 'X' ) | X */
/*   */

BDD_STATUS bdd_and (Bdd1, Bdd2, BddRes)
{
    BDD      Bdd1;
    BDD      Bdd2;
    BDD      *BddRes;
{
    BDD      Res;
    BDD_STATUS Status;

    if ((Bdd1 == BDD_0) || (Bdd2 == BDD_0))
    {
        *BddRes = BDD_0;
        return (BDD_OK);
    }

    if ((Bdd1 == BDD_Z) || (Bdd2 == BDD_Z))
    {
        *BddRes = BDD_X;
        return (BDD_OK);
    }

    if ((Bdd1 == BDD_X) || (Bdd2 == BDD_X))

```

```

    {
        *BddRes = BDD_X;
        return (BDD_OK);
    }

if (Bdd1 == BDD_1)
{
    /* If no Z in "Bdd2" then we can directly return otherwise */
    /* we need to continue to solve each leaves (this is due to the */
    /* fact that (1.Z) -> X (and not Z). */
    if (BDD_Z_NOT_USED)
    {
        *BddRes = UseBdd (Bdd2);
        return (BDD_OK);
    }
}

if (Bdd2 == BDD_1)
{
    /* If no Z in "Bdd1" then we can directly return otherwise */
    /* we need to continue to solve each leaves (this is due to the */
    /* fact that (1.Z) -> X (and not Z). */
    if (BDD_Z_NOT_USED)
    {
        *BddRes = UseBdd (Bdd1);
        return (BDD_OK);
    }
}

if (IsBddTypeINV (Bdd1))
{
    if (IsBddTypeINV (Bdd2))
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_or,
                                      GetBddPtrFromBdd (Bdd1),
                                      GetBddPtrFromBdd (Bdd2),
                                      GLOB_BDD_WS->OrCache,
                                      &Res)))
            return (Status);

        *BddRes = ComplementBDD(Res);
        return (BDD_OK);
    }
    else
    {
        if (GetBddPtrFromBdd (Bdd1) == (BDD_PTR)Bdd2)
        {
            *BddRes = BDD_0;
            return (BDD_OK);
        }
    }
}

```

bbdd.c

```
if ((Status = ApplyNoCommutOp (bdd_impl,
                               (BDD_PTR) Bdd2,
                               GetBddPtrFromBdd (Bdd1),
                               GLOB_BDD_WS->ImpCache,
                               &Res)))
    return (Status);

*BddRes = ComplementBDD(Res);
return (BDD_OK);
}
}
else
{
    if (IsBddTypeINV (Bdd2))
    {
        if ((BDD_PTR) Bdd1 == GetBddPtr(Bdd2))
        {
            *BddRes = BDD_0;
            return (BDD_OK);
        }
        if ((Status = ApplyNoCommutOp (bdd_impl,
                                       (BDD_PTR) Bdd1,
                                       GetBddPtr (Bdd2),
                                       GLOB_BDD_WS->ImpCache,
                                       &Res)))
            return (Status);

        *BddRes = ComplementBDD(Res);
        return (BDD_OK);
    }
    else
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_and,
                                     (BDD_PTR) Bdd1,
                                     (BDD_PTR) Bdd2,
                                     GLOB_BDD_WS->AndCache,
                                     BddRes)))
            return (Status);

        return (BDD_OK);
    }
}

/*
/* bdd_nand
/*
/* Logical Operator NAND */
/*
/*-----*/
```

bbdd.c

```

BDD_STATUS bdd_nand (Bdd1, Bdd2, BddRes)
    BDD Bdd1;
    BDD Bdd2;
    BDD *BddRes;
{
    BDD             Res;
    BDD_STATUS      Status;

    if ((Status = bdd_and (Bdd1, Bdd2, &Res)))
        return (Status);

    *BddRes = ComplementBDD(Res);
    return (BDD_OK);
}                                /* bdd_nand */

/*
/* bdd_or
/*
/* Logical Operator OR
/*
/* -- truth table for "or" function
/* CONSTANT or_table : stdlogic_table := (
/*
-- | U   X   0   1   Z   W   L   H   -   |   |
/*
-- |   |   |   |   |   |   |   |   |   |   |
/*
-- |   ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ), -- | U | */
-- |   ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | X | */
-- |   ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 | */
-- |   ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | 1 | */
-- |   ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | Z | */
-- |   ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | W | */
-- |   ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L | */
-- |   ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | H | */
-- |   ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ) -- | - | */
/*
/* For 4 values (1, 0, Z, X) we extract:
/*
-- | 1   0   Z   X   |   |
/*
-- |   |   |   |   |
/*
-- |   ( '1', '1', '1', '1' ) | 1 | */
-- |   ( '1', '0', 'X', 'X' ) | 0 | */
-- |   ( '1', 'X', 'X', 'X' ) | Z | */
-- |   ( '1', 'X', 'X', 'X' ) | X | */
/*
BDD_STATUS bdd_or (Bdd1, Bdd2, BddRes)
    BDD     Bdd1;
    BDD     Bdd2;
    BDD     *BddRes;
{
    BDD             Res;
    BDD_STATUS      Status;

    if ((Bdd1 == BDD_1) || (Bdd2 == BDD_1))

```

```

{
    *BddRes = BDD_1;
    return (BDD_OK);
}

if ((Bdd1 == BDD_Z) || (Bdd2 == BDD_Z))
{
    *BddRes = BDD_X;
    return (BDD_OK);
}

if ((Bdd1 == BDD_X) || (Bdd2 == BDD_X))
{
    *BddRes = BDD_X;
    return (BDD_OK);
}

if (Bdd1 == BDD_0)
{
    /* If no Z in "Bdd2" then we can directly return otherwise */
    /* we need to continue to solve each leaves (this is due to the */
    /* fact that (0+Z) -> X (and not Z)). */
    if (BDD_Z_NOT_USED)
    {
        *BddRes = UseBdd (Bdd2);
        return (BDD_OK);
    }
}

if (Bdd2 == BDD_0)
{
    /* If no Z in "Bdd1" then we can directly return otherwise */
    /* we need to continue to solve each leaves (this is due to the */
    /* fact that (0+Z) -> X (and not Z)). */
    if (BDD_Z_NOT_USED)
    {
        *BddRes = UseBdd (Bdd1);
        return (BDD_OK);
    }
}

if (IsBddTypeINV (Bdd1))
{
    if (IsBddTypeINV (Bdd2))
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_and,
                                     GetBddPtr (Bdd1),
                                     GetBddPtr (Bdd2),
                                     GLOB_BDD_WS->AndCache,
                                     &Res)))
            return (Status);
    }
}

```

```

        *BddRes = ComplementBDD(Res);
        return (BDD_OK);
    }
else
{
    if (GetBddPtr (Bdd1) == (BDD_PTR)Bdd2)
    {
        *BddRes = BDD_1;
        return (BDD_OK);
    }
    if ((Status = ApplyNoCommutOp (bdd_impl,
                                    GetBddPtr (Bdd1),
                                    (BDD_PTR)Bdd2,
                                    GLOB_BDD_WS->ImpCache,
                                    BddRes)))
        return (Status);

    return (BDD_OK);
}
else
{
    if (IsBddTypeINV (Bdd2))
    {
        if ((BDD_PTR)Bdd1 == GetBddPtr(Bdd2))
        {
            *BddRes = BDD_1;
            return (BDD_OK);
        }
        if ((Status = ApplyNoCommutOp (bdd_impl,
                                        GetBddPtr (Bdd2),
                                        (BDD_PTR)Bdd1,
                                        GLOB_BDD_WS->ImpCache,
                                        BddRes)))
            return (Status);

        return (BDD_OK);
    }
else
{
    if (Bdd1 == Bdd2)
    {
        *BddRes = UseBdd (Bdd2);
        return (BDD_OK);
    }
    if ((Status = ApplyCommutOp (bdd_or,
                                (BDD_PTR)Bdd1,
                                (BDD_PTR)Bdd2,
                                GLOB_BDD_WS->OrCache,
                                BddRes)))
        return (Status);

    return (BDD_OK);
}
}

```

bbdd.c

```
}

/*-----*/
/* bdd_nor */
/*-----*/
/* Logical Operator NOR. */
/*-----*/
BDD_STATUS bdd_nor (Bdd1, Bdd2, BddRes)
    BDD      Bdd1;
    BDD      Bdd2;
    BDD      *BddRes;
{
    BDD          Res;
    BDD_STATUS   Status;

    if ((Status = bdd_or (Bdd1, Bdd2, &Res)))
        return (Status);

    *BddRes = ComplementBDD(Res);
    return (BDD_OK);
} /* bdd_nor */

/*-----*/
/* bdd_not */
/*-----*/
/* Logical Operator NOT. */
/*-----*/
/* -- truth table for "not" function */
/* CONSTANT not_table: stdlogic_1d := */
/* -- ----- */
/* | U X 0 1 Z W L H - | */
/* -- ----- */
/* ('U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ); */
/* For 4 values (1, 0, Z, X) we extract: */
/* | 1 0 Z X | */
/* ----- */
/* ('0', '1', 'X', 'X' ) */
/*-----*/
BDD_STATUS bdd_not (Bdd, BddRes)
    BDD      Bdd;
    BDD      *BddRes;

{
    /* Treatment od BDD_Z and BDD_X are done in macro "ComplementBDD" */
    *BddRes = ComplementBDD(UseBdd (Bdd));
    return (BDD_OK);
} /* bdd_not */

/*-----*/
/* bdd_impl */
/*-----*/
```

bbdd.c

```
/*-----*/
/* Logical Operator Impl (Bdd1 => Bdd2) which comes down to be          */
/* (! Bdd1 + Bdd2).                                                       */
/*
/* For 4 values (1, 0, Z, X) we extract:                                     */
/*
/*   Bdd1 | 1  0  Z  X  | Bdd2 |                                         */
/*   -----|-----|-----|-----|
/*   ('1', '1', '1', '1') | 1 |
/*   ('0', '1', 'X', 'X') | 0 |
/*   ('X', '1', 'X', 'X') | Z |
/*   ('X', '1', 'X', 'X') | X |
/*
/*-----*/
BDD_STATUS bdd_impl (Bdd1, Bdd2, BddRes)
    BDD      Bdd1;
    BDD      Bdd2;
    BDD      *BddRes;
{
    BDD          Res;
    BDD_STATUS   Status;

    if (Bdd1 == BDD_0)
    {
        *BddRes = BDD_1;
        return (BDD_OK);
    }

    if (Bdd2 == BDD_1)
    {
        *BddRes = BDD_1;
        return (BDD_OK);
    }

    if ((Bdd1 == BDD_Z) || (Bdd2 == BDD_Z))
    {
        *BddRes = BDD_X;
        return (BDD_OK);
    }

    if ((Bdd1 == BDD_X) || (Bdd2 == BDD_X))
    {
        *BddRes = BDD_X;
        return (BDD_OK);
    }

    if (Bdd1 == BDD_1)
    {
        /* If no Z in "Bdd2" then we can directly return otherwise */
        /* we need to continue to solve each leaves (this is due to the */
        /* fact that (!l+Z) -> X (and not Z)).                         */
        if (BDD_Z_NOT_USED)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
    }
}
```

```

}

if (Bdd2 == BDD_0)
{
    /* No problem here is there is Z in "Bdd1" because on the Not      */
    /* which makes Z into X (and X+0 -> X)                                */
    *BddRes = UseBdd (ComplementBDD(Bdd1));
    return (BDD_OK);
}

if (IsBddTypeINV (Bdd1))
{
    if (IsBddTypeINV (Bdd2))
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = BDD_1;
            return (BDD_OK);
        }
        if ((Status = ApplyNoCommutOp (bdd_impl,
                                         GetBddPtr (Bdd2),
                                         GetBddPtr (Bdd1),
                                         GLOB_BDD_WS->ImpCache,
                                         BddRes)))
            return (Status);

        return (BDD_OK);
    }
    else
    {
        if (GetBddPtr (Bdd1) == (BDD_PTR)Bdd2)
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_or,
                                     GetBddPtr (Bdd1),
                                     (BDD_PTR)Bdd2,
                                     GLOB_BDD_WS->OrCache,
                                     BddRes)))
            return (Status);

        return (BDD_OK);
    }
}
else
{
    if (IsBddTypeINV (Bdd2))
    {
        if ((BDD_PTR)Bdd1 == GetBddPtr (Bdd2))
        {
            *BddRes = UseBdd (Bdd2);
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_and,

```

```

        (BDD_PTR)Bdd1,
        GetBddPtr (Bdd2),
        GLOB_BDD_WS->AndCache,
        &Res)))
    return (Status);

    *BddRes = ComplementBDD(Res);
    return (BDD_OK);
}
else
{
    if (Bdd1 == Bdd2)
    {
        *BddRes = BDD_1;
        return (BDD_OK);
    }
    if ((Status = ApplyNoCommutOp (bdd_impl,
                                    (BDD_PTR)Bdd1,
                                    (BDD_PTR)Bdd2,
                                    GLOB_BDD_WS->ImpCache,
                                    BddRes))) != BDD_OK)
        return (Status);

    return (BDD_OK);
}
}

/*
 * bdd_equi
 */
/*
 * Logical Operator EQUI (Bdd1 <=> Bdd2) which is the NXor: */
/* (Bdd1 . Bdd2 + !Bdd1 . !Bdd2) */
/*
 * -- truth table for "xor" function
/* CONSTANT xor_table : stdlogic_table := (
/*   -----
/*   | U X 0 1 Z W L H - | | */
/*   ----- */
/*   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U */
/*   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X */
/*   ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 */
/*   ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | 1 */
/*   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | Z */
/*   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | W */
/*   ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L */
/*   ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | H */
/*   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - */

/*
 * For 4 values (X, 0, 1, Z) for the XNor we extract:
*/
/* Bdd1 | X 0 1 Z | Bdd2 | */
/* ----- */
/* ( 'X', 'X', 'X', 'X' ) | X | */

```

bbdd.c

```
/*
 *      ( 'X', '1', '0', 'X' ) |  0  |
 *      ( 'X', '0', '1', 'X' ) |  1  |
 *      ( 'X', 'X', 'X', 'X' ) |  Z  |
 */
/*-----*/
BDD_STATUS bdd_equi (Bdd1, Bdd2, BddRes)
    BDD     Bdd1;
    BDD     Bdd2;
    BDD     *BddRes;
{
    BDD         Res;
    BDD_STATUS   Status;

    if ((Bdd1 == BDD_X) || (Bdd2 == BDD_X))
    {
        *BddRes = BDD_X;
        return (BDD_OK);
    }

    if ((Bdd1 == BDD_Z) || (Bdd2 == BDD_Z))
    {
        *BddRes = BDD_Z;
        return (BDD_OK);
    }

    if (Bdd1 == BDD_1)
    {
        *BddRes = UseBdd (Bdd2);
        return (BDD_OK);
    }

    if (Bdd1 == BDD_0)
    {
        *BddRes = UseBdd (ComplementBDD(Bdd2));
        return (BDD_OK);
    }

    if (Bdd2 == BDD_0)
    {
        *BddRes = UseBdd (ComplementBDD(Bdd1));
        return (BDD_OK);
    }

    if (Bdd2 == BDD_1)
    {
        /* If no Z in "Bdd1" then we can directly return otherwise */
        /* we need to continue to solve each leaves (this is due to the */
        /* fact that (Z<=>1) -> X (and not Z)). */
        if (BDD_Z_NOT_USED)
        {
            *BddRes = UseBdd(Bdd1);
            return (BDD_OK);
        }
    }
}
```

```

if (IsBddTypeINV (Bdd1))
{
    if (IsBddTypeINV (Bdd2))
    {
        if (Bdd1 == Bdd2)
        {
            *BddRes = BDD_1;
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_equi,
                                      GetBddPtr (Bdd2),
                                      GetBddPtr (Bdd1),
                                      GLOB_BDD_WS->EqCache,
                                      BddRes)))
            return (Status);

        return (BDD_OK);
    }
else
{
    if (GetBddPtr (Bdd1) == (BDD_PTR) Bdd2)
    {
        *BddRes = BDD_0;
        return (BDD_OK);
    }
    if ((Status = ApplyCommutOp (bdd_equi,
                                 GetBddPtr (Bdd1),
                                 (BDD_PTR) Bdd2,
                                 GLOB_BDD_WS->EqCache,
                                 &Res)))
        return (Status);

    *BddRes = ComplementBDD(Res);
    return (BDD_OK);
}
else
{
    if (IsBddTypeINV (Bdd2))
    {
        if ((BDD_PTR) Bdd1 == GetBddPtr(Bdd2))
        {
            *BddRes = BDD_0;
            return (BDD_OK);
        }
        if ((Status = ApplyCommutOp (bdd_equi,
                                     (BDD_PTR) Bdd1,
                                     GetBddPtr (Bdd2),
                                     GLOB_BDD_WS->EqCache,
                                     &Res)))
            return (Status);

        *BddRes = ComplementBDD(Res);
        return (BDD_OK);
    }
}

```

bbdd.c

```
if (Bdd1 == Bdd2)
{
    *BddRes = BDD_1;
    return (BDD_OK);
}
if ((Status = ApplyCommutOp (bdd_equi,
                             (BDD_PTR)Bdd1,
                             (BDD_PTR)Bdd2,
                             GLOB_BDD_WS->EqCache,
                             BddRes)))
    return (Status);

return (BDD_OK);
}
}

/*
/* bdd_xor
 */
/*
/* Boolean XOR function applied.
*/
BDD_STATUS bdd_xor (Bdd1, Bdd2, BddRes)
BDD      Bdd1;
BDD      Bdd2;
BDD      *BddRes;
{
    BDD          Res;
    BDD_STATUS   Status;

    if ((Status = bdd_equi (ComplementBDD (Bdd1), Bdd2, BddRes)))
        return (Status);

    return (BDD_OK);
}

/*
/* ResetTagOccurBitInWs
*/
/*
/* Reset all the Occur bits of all the bdd nodes in the work space "WS".*/
/* This is a brutal but safe reset.
*/
void ResetTagOccurBitInWs (Ws)
    BDD_WS    Ws;
{
    int          Index;
    int          i;
    BDD_PTR    BddPtr;
    BLIST      Cache;
    int          NbVar;

    Cache = Ws->BddCache;
```

```

NbVar = BListSize (Cache);

for (Index = 0; Index < NbVar; Index++)
{
    /* If at least one node in this bucket we need to scan it.      */
    if (GetCache (Cache, Index) ->NbBddNode)
    {
        for (i=0; i<GetCache (Cache, Index) ->CacheSize; i++)
        {
            BddPtr = GetCache (Cache, Index) ->Buckets[i];
            while (BddPtr != NULL)
            {
                ResetBddTagOccurBit (BddPtr);
                BddPtr = BddPtr->Next;
            }
        }
    }
}

/*-----*/
/* ResetTagOccurBit                               */
/*-----*/
void ResetTagOccurBit (Bdd)
    BDD      Bdd;
{
    BDD_PTR  BddPtr;

    if (ConstantBdd (Bdd))
        return;

    BddPtr = GetBddPtrFromBdd (Bdd);

    if (!IsBddTagOccurBit (BddPtr))
        return;

    ResetBddTagOccurBit (BddPtr);

    ResetTagOccurBit (GetBddPtrEdge1 (BddPtr));
    ResetTagOccurBit (GetBddPtrEdge0 (BddPtr));
}

/*-----*/
/* ResetTagCountBit                             */
/*-----*/
void ResetTagCountBit (Bdd)
    BDD      Bdd;
{
    BDD_PTR  BddPtr;

    if (ConstantBdd (Bdd))

```

bbdd.c

```
    return;

    BddPtr = GetBddPtrFromBdd (Bdd);

    if (!IsBddTagCountBit (BddPtr))
        return;

    ResetBddTagCountBit (BddPtr);

    ResetTagCountBit (GetBddPtrEdge1 (BddPtr));
    ResetTagCountBit (GetBddPtrEdge0 (BddPtr));

}

/*-----*/
/* bdd_compose_rec */
/*-----*/
/* Substitutes a list of Variables by their associated BDD. The list Var*/
/* must be sorted according to the field ->Id in the increasing order. */
/* */
/* WARNING!: this function may involves troubles if use conjointly with */
/* functions using the "TagOccurBit" (ex: BddCountNodesAndMark). */
/*-----*/
BDD_STATUS bdd_compose_rec (Bdd, VarIndex, BddSub, BddRes)
    BDD          Bdd;
    Uint32      VarIndex;
    BDD          BddSub;
    BDD          *BddRes;
{
    BDD_PTR    BddPtr;
    Uint32      index;
    BDD_STATUS   Status;
    BDD          Res;

    if (ConstantBdd (Bdd))
    {
        *BddRes = Bdd;
        return (BDD_OK);
    }

    BddPtr = GetBddPtrFromBdd (Bdd);

    /* Checking TagOccurBit to reuse eventually previous computation. */
    if (IsBddTagOccurBit (BddPtr))
    {
        if (BddPtr != (BDD_PTR)Bdd)
            *BddRes = UseBdd (ComplementBDD (GetBddPtrCacheCompose (BddPtr)));
        else
            *BddRes = UseBdd (GetBddPtrCacheCompose (BddPtr));

        return (BDD_OK);
    }

    index = GetBddPtrIndex (BddPtr);
```

```

/* No greater index so the result is BddPtr itself. */
if ((GLOB_BDD_WS->IndexVarOrder && (index < VarIndex)) ||
    (! (GLOB_BDD_WS->IndexVarOrder) && (index > VarIndex)))
{
    if (BddPtr != (BDD_PTR)Bdd)
        *BddRes = UseBdd (ComplementBDD((BDD)BddPtr));
    else
        *BddRes = UseBdd ((BDD)BddPtr);

    return (BDD_OK);
}

switch ((GLOB_BDD_WS->CompareIdx) (index, VarIndex))
{
    BDD          AndL;
    BDD          AndR;
    BDD          Bdd1;
    BDD          Bdd0;
    BDD          Self;

case -1:
    BddTagOccurBit (BddPtr);

    if ((Status = bdd_compose_rec (GetBddPtrEdge1 (BddPtr),
                                   VarIndex, BddSub, &Bdd1)))
        return (Status);

    if ((Status = bdd_compose_rec (GetBddPtrEdge0 (BddPtr),
                                   VarIndex, BddSub, &Bdd0)))
        return (Status);

    if ((Bdd1 == GetBddPtrEdge1 (BddPtr)) &&
        (Bdd0 == GetBddPtrEdge0 (BddPtr)))
    {
        FreeBdd (Bdd1);
        FreeBdd (Bdd0);
        SetBddPtrCacheCompose (BddPtr, (BDD)UseBddPtr (BddPtr));
        Res = GetBddPtrCacheCompose (BddPtr);
        if (BddPtr != (BDD_PTR)Bdd)
            *BddRes = UseBdd (ComplementBDD(Res));
        else
            *BddRes = UseBdd (Res);

        return (BDD_OK);
    }

    Self = UseBdd (GetBddVar(index)->Bdd);

    if ((Status = bdd_and (Bdd0, ComplementBDD (Self), &AndL)))
        return (Status);
    FreeBdd (Bdd0);

    if ((Status = bdd_and (Bdd1, Self, &AndR)))
        return (Status);
    FreeBdd (Bdd1);
}

```

```

        if ((Status = bdd_or (AndL, AndR, &Res)))
            return (Status);
        SetBddPtrCacheCompose (BddPtr, UseBdd (Res));
        FreeBdd (AndL);
        FreeBdd (AndR);
        FreeBdd (Self);

        break;

    case 0:
        if ((Status = bdd_and (GetBddPtrEdge0 (BddPtr),
                               ComplementBDD (BddSub), &AndL)))
            return (Status);

        if ((Status = bdd_and (GetBddPtrEdge1 (BddPtr),
                               BddSub, &AndR)))
            return (Status);

        if ((Status = bdd_or (AndL, AndR, &Res)))
            return (Status);
        SetBddPtrCacheCompose (BddPtr, UseBdd (Res));
        BddTagOccurBit (BddPtr);
        FreeBdd (AndL);
        FreeBdd (AndR);
    }

    if (BddPtr != (BDD_PTR)Bdd)
        *BddRes = UseBdd (ComplementBDD(Res));
    else
        *BddRes = UseBdd (Res);

    return (BDD_OK);
}

/*
 *-----*
 * bdd_compose
 *-----*/
/*
 * Substitutes variable of index "index" with bdd "BddSub" in the bdd
 * "Bdd".
 */
/*
 * PREC!: The tag occur bit must be set to 0 for all the bdd nodes of
 * "Bdd".
 */
/*
 *-----*
 BDD_STATUS bdd_compose (Bdd, index, BddSub, BddRes)
{
    BDD      Bdd;
    int      index;
    BDD      BddSub;
    BDD      *BddRes;
{
    BDD_STATUS      Status;

    Status = bdd_compose_rec (Bdd, index, BddSub, BddRes);
    (void)ResetTagOccurBit (Bdd);
}

```

```

        return (Status);
    }

/*-----*/
/* bdd_cofac0                                     */
/*-----*/
/* Builds the cofactor 0 of variable index "index" of Bdd "Bdd". */
/*-----*/
BDD_STATUS bdd_cofac0 (Bdd, index, BddRes)
    BDD      Bdd;
    int      index;
    BDD      *BddRes;
{
    BDD_STATUS      Status;

    Status = bdd_compose_rec (Bdd, index, bdd_zero (), BddRes);
    (void)ResetTagOccurBit (Bdd);

    return (Status);
}

/*-----*/
/* bdd_cofac1                                     */
/*-----*/
/* Builds the cofactor 1 of variable index "index" of Bdd "Bdd". */
/*-----*/
BDD_STATUS bdd_cofac1 (Bdd, index, BddRes)
    BDD      Bdd;
    int      index;
    BDD      *BddRes;
{
    BDD_STATUS      Status;

    Status = bdd_compose_rec (Bdd, index, bdd_one (), BddRes);
    (void)ResetTagOccurBit (Bdd);

    return (Status);
}

/*-----*/
/* bdd_restrict                                    */
/*-----*/
/* Logical Operator bdd_restrict the restrict operator with tdg as the */
/* Bdd1 and dmn as the chi function of the domain. */
/*-----*/
BDD_STATUS bdd_restrict (Bdd1, dmn, BddRes)
    BDD      Bdd1;
    BDD      dmn;
    BDD      *BddRes;
{
    BDD          Res;
    BDD_STATUS    Status;

    if ((Bdd1 == BDD_X) || (dmn == BDD_X))
    {

```

bbdd.c

```
*BddRes = BDD_X;
return (BDD_OK);
}

if (dmn == BDD_0)
{
    *BddRes = BDD_0;
    return (BDD_OK);
}

if (ConstantBdd (Bdd1))
{
    *BddRes = Bdd1;
    return (BDD_OK);
}

if ((dmn == BDD_1) || (dmn == BDD_Z))
{
    *BddRes = UseBdd (Bdd1);
    return (BDD_OK);
}

if (Bdd1 == dmn)
{
    *BddRes = BDD_1;
    return (BDD_OK);
}

if (GetBddPtr(Bdd1) == GetBddPtr(dmn))
{
    *BddRes = BDD_0;
    return (BDD_OK);
}

if (IsBddTypeINV (Bdd1))
{
    if ((Status = ApplyRestrict (GetBddPtrFromBdd (Bdd1), dmn,
                                GLOB_BDD_WS->RstCache, &Res)))
        return (Status);

    *BddRes = ComplementBDD(Res);
    return (BDD_OK);
}
else
{
    if ((Status = ApplyRestrict ((BDD_PTR)Bdd1, dmn,
                                GLOB_BDD_WS->RstCache, BddRes)))
        return (Status);

    return (BDD_OK);
}

}

/*-----
/* BddCountNodesRec
 */
```

bbdd.c

```

/*
 * CAUTION!: Field Hook is modified so take care if you use it outside !*/
/* The bit 0 is modified */
*/
static int BddCountNodesRec (Bdd)
    BDD Bdd;
{
    BDD_PTR BddPtr;

    if (ConstantBdd (Bdd))
        return (0);

    BddPtr = GetBddPtrFromBdd (Bdd);

    if (IsBddTagCountBit (BddPtr))
        return (0);

    BddTagCountBit (BddPtr);

    return (1 +
        BddCountNodesRec (GetBddPtrEdge1 (BddPtr)) +
        BddCountNodesRec (GetBddPtrEdge0 (BddPtr)));
}

/*
 * BddCountNodesAndMark
 */
/*
 * CAUTION!: two LSB bits of Field Hook can be modified so take care if *
 * you use this field !
 * The nodes are marked thru the Field Hook so if you rerun the same      */
 * function it will not count the previous visited nodes.          */
 * This function updates also Field ->TotalCountBddNode of the current   */
 * BDD work space.                                              */
void BddCountNodesAndMark (Bdd, NbNode)
    BDD      Bdd;
    int     *NbNode;
{
    *NbNode = BddCountNodesRec (Bdd);

    GLOB_BDD_WS->TotalCountBddNode += *NbNode;
}

/*
 * BddCountNodes
 */
/*
 * Counts the exact number of nodes of the bdd "Bdd" but removes marks   */
 * after traversal in order to do not have side effects on next call to */
 * this function.                                              */
void BddCountNodes (Bdd, NbNode)
    BDD      Bdd;
    int     *NbNode;

```

bbdd.c

```
{  
    *NbNode = BddCountNodesRec (Bdd);  
    (void)ResetTagCountBit (Bdd);  
}  
  
/*-----*/  
/* BddCountNodesOnVarRec */  
/*-----*/  
/* CAUTION!: Field Hook is modified so take care if you use it outside !*/  
/* The bit 0 is modified */  
/*-----*/  
static int BddCountNodesOnVarRec (Bdd)  
    BDD          Bdd;  
{  
    BDD_PTR    BddPtr;  
    BDD_VAR    Var;  
  
    if (ConstantBdd (Bdd))  
        return (0);  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    if (IsBddTagCountBit (BddPtr))  
        return (0);  
  
    Var = GetBddVar (GetBddPtrIndex (BddPtr));  
  
    BddTagCountBit (BddPtr);  
  
    (Var->Hook)++;  
  
    return (1 +  
            BddCountNodesOnVarRec (GetBddPtrEdge1 (BddPtr)) +  
            BddCountNodesOnVarRec (GetBddPtrEdge0 (BddPtr)));  
}  
  
/*-----*/  
/* BddCountNodesOnVar */  
/*-----*/  
void BddCountNodesOnVar (Bdd, NbNode)  
    BDD          Bdd;  
    int         *NbNode;  
{  
  
    *NbNode = BddCountNodesOnVarRec (Bdd);  
    (void)ResetTagCountBit (Bdd);  
}  
  
/*-----*/  
/* UpDateBddRec */  
/*-----*/  
void UpDateBddRec (Bdd, VarToSub)  
    BDD          Bdd;  
    BDD_VAR    VarToSub;
```

bbdd.c

```
{  
    BDD_PTR BddPtr;  
  
    if (ConstantBdd (Bdd))  
        return;  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    if (IsBddTagOccurBit (BddPtr))  
        return;  
  
    if (GetBddPtrIndex (BddPtr) < VarToSub->Id)  
        SetBddPtrIndex (BddPtr, GetBddPtrIndex (BddPtr)+1);  
    else  
        return;  
  
    BddTagOccurBit (BddPtr);  
  
    UpDateBddRec (GetBddPtrEdge1 (BddPtr), VarToSub);  
    UpDateBddRec (GetBddPtrEdge0 (BddPtr), VarToSub);  
  
}  
  
/*-----*/  
/* UpDateBdd */  
/*-----*/  
void UpDateBdd (Bdd, VarToSub)  
{  
    BDD Bdd;  
    BDD_VAR VarToSub;  
    int i;  
    BDD_PTR BddPtr;  
  
    for (i=1; i<BListSize (GLOB_BDD_WS->Var); i++)  
    {  
        if (GetBddVar(i)->Id < VarToSub->Id)  
            (GetBddVar(i)->Id)++;  
    }  
  
    VarToSub->Id = 1;  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    SetBddPtrIndex (BddPtr, 1);  
  
    UpDateBddRec (GetBddPtrEdge1 (BddPtr), VarToSub);  
    UpDateBddRec (GetBddPtrEdge0 (BddPtr), VarToSub);  
  
    (void)ResetTagOccurBit (Bdd);  
}  
  
/*-----*/  
/* GetNbNode */  
/*-----*/
```

bbdd.c

```
/*-----*/
int GetNbNode (Bdd, j)
    BDD      Bdd;
    int      j;
{
    int          NbNode;
    BDD_PTR  BddPtr;
    BDD_PTR  BddPtr_1;
    BDD_PTR  BddPtr_0;

    BddPtr = GetBddPtrFromBdd (Bdd);
    if (GetBddPtrIndex (BddPtr) == j)
        return (1);

    BddPtr_1 = GetBddPtrFromBdd (GetBddPtrEdge1 (BddPtr));
    BddPtr_0 = GetBddPtrFromBdd (GetBddPtrEdge0 (BddPtr));

    NbNode = 1;
    if (GetBddPtrIndex (BddPtr_1) == j)
        NbNode++;
    if (GetBddPtrIndex (BddPtr_0) == j)
        NbNode++;

    return (NbNode);
}

/*-----*/
/* bdd_swap_node */
/*-----*/
/*
/*      Description:      swap adjacent indices in one BDD.
/*
/*
/*      x
/*      / \
/*      Y   Y
/*      / \ / \
/*      a   b   c   d      =>      y
/*                           / \
/*                           x   x
/*                           / \ / \
/*                           a   c   b   d
/*
/*
/*      x
/*      / \
/*      Y   \
/*      / \   \
/*      a   b   c      =>      y
/*                           / \
/*                           x   x
/*                           / \ / \
/*                           a   c   b   c
/*
/*
/*      x
/*      / \
/*      /   Y
/*      /   / \
/*      a   c   d      =>      y
/*                           / \
/*                           x   x
/*                           / \ / \
/*                           a   c   a   d
/*
/*
/*      z
/*      / \
/*      Y   Y
/*      / \ / \
/*      / \ / \      =>      z
/*      / \ / \      / \
/*      x   x   x   x
```

bbdd.c

```
/*
 *      a   b   c   d           a   c   b   d           */
/*          x                   Y
 *          / \                 / \
 *          a   c               a   c
 *          =>                  */
/*          y                   x
 *          / \                 / \
 *          a   c               a   c
 *          =>                  */
/*          */
/*-----*/
BDD_STATUS bdd_swap_node (Bdd, i, j, GainNbNode, BddRes)
    BDD      Bdd;
    int     i;
    int     j;
    int     *GainNbNode;
    BDD     *BddRes;
{
    BDD_PTR  BddPtr;
    Uint32   IndexBdd;
    BDD      MinBddLocal;
    BDD_STATUS Status;
    int      NbNode1;
    int      NbNode2;

    BddPtr = GetBddPtrFromBdd (Bdd);
    IndexBdd = GetBddPtrIndex (BddPtr);

    if (IndexBdd == i)
    {
        BDD_PTR  BddPtr_1 = GetBddPtrFromBdd (GetBddPtrEdge1 (BddPtr));
        BDD_PTR  BddPtr_0 = GetBddPtrFromBdd (GetBddPtrEdge0 (BddPtr));
        Uint32   IndexBdd_1 = GetBddPtrIndex (BddPtr_1);
        Uint32   IndexBdd_0 = GetBddPtrIndex (BddPtr_0);
        BDD     Bdd_a;
        BDD     Bdd_b;
        BDD     Bdd_c;
        BDD     Bdd_d;
        BDD     Bdd1;
        BDD     Bdd2;
        int     TypeBdd;

        if (IndexBdd_1 == j)
        {
            if (IndexBdd_0 == j)
            {
                Bdd_a = UseBdd (GetBddPtrEdge0 (BddPtr_0));
                Bdd_b = UseBdd (GetBddPtrEdge1 (BddPtr_0));
                Bdd_c = UseBdd (GetBddPtrEdge0 (BddPtr_1));
                Bdd_d = UseBdd (GetBddPtrEdge1 (BddPtr_1));
                TypeBdd = IsBddTypeINV (GetBddPtrEdge0 (BddPtr));
            }
            else
            {
                Bdd_a = UseBdd (GetBddPtrEdge0 (BddPtr));
            }
        }
    }
}
```

```

        Bdd_b = UseBdd (GetBddPtrEdge0 (BddPtr));
        Bdd_c = UseBdd (GetBddPtrEdge0 (BddPtr_1));
        Bdd_d = UseBdd (GetBddPtrEdge1 (BddPtr_1));
        TypeBdd = 0;
    }
}
else
{
    if (IndexBdd_0 == j)
    {
        Bdd_a = UseBdd (GetBddPtrEdge0 (BddPtr_0));
        Bdd_b = UseBdd (GetBddPtrEdge1 (BddPtr_0));
        Bdd_c = UseBdd (GetBddPtrEdge1 (BddPtr));
        Bdd_d = UseBdd (GetBddPtrEdge1 (BddPtr));
        TypeBdd = IsBddTypeINV (GetBddPtrEdge0 (BddPtr));
    }
    else
    {
        Bdd_a = UseBdd (GetBddPtrEdge0 (BddPtr));
        Bdd_b = UseBdd (Bdd_a);
        Bdd_c = UseBdd (GetBddPtrEdge1 (BddPtr));
        Bdd_d = UseBdd (Bdd_c);
        TypeBdd = 0;
    }
}

if (TypeBdd)
{
    if ((Status = MakeBdd (j, ComplementBDD (Bdd_a), Bdd_c, &Bdd1)))
        return (Status);
    if ((Status = MakeBdd (j, ComplementBDD (Bdd_b), Bdd_d, &Bdd2)))
        return (Status);
    if ((Status = MakeBdd (i, Bdd1, Bdd2, &MinBddLocal)))
        return (Status);
    FreeBdd (Bdd1);
    FreeBdd (Bdd2);
}
else
{
    if ((Status = MakeBdd (j, Bdd_a, Bdd_c, &Bdd1)))
        return (Status);
    if ((Status = MakeBdd (j, Bdd_b, Bdd_d, &Bdd2)))
        return (Status);
    if ((Status = MakeBdd (i, Bdd1, Bdd2, &MinBddLocal)))
        return (Status);
    FreeBdd (Bdd1);
    FreeBdd (Bdd2);
}
}
else if (IndexBdd == j)
{
    BDD Bdd_a;
    BDD Bdd_b;

    Bdd_a = (BDD) UseBddPtr (GetBddPtrFromBdd (GetBddPtrEdge0 (BddPtr)));
    Bdd_b = (BDD) UseBddPtr (GetBddPtrFromBdd (GetBddPtrEdge1 (BddPtr)));
}

```

bbdd.c

```
if (IsBddTypeINV (GetBddPtrEdge0 (BddPtr)))
{
    if ((Status = MakeBdd (i, ComplementBDD (Bdd_a), Bdd_b,
                           &MinBddLocal)))
        return (Status);
    }
else
{
    if ((Status = MakeBdd (i, Bdd_a, Bdd_b, &MinBddLocal)))
        return (Status);
    }
}
else
{
    if (IsBddTypeINV (Bdd))
    {
        if ((Status = bdd_not (Bdd, &MinBddLocal)))
            return (Status);
        }
    else
        MinBddLocal = UseBdd (Bdd);
    }

if (IsBddTypeINV (Bdd))
    (*BddRes) = ComplementBDD (MinBddLocal);
else
    (*BddRes) = MinBddLocal;

*GainNbNode = 0;

if (Bdd == *BddRes)
    return (BDD_OK);

NbNode1 = GetNbNode (Bdd, j);
NbNode2 = GetNbNode (*BddRes, j);
*GainNbNode = NbNode1-NbNode2;

return (BDD_OK);
}

/*-----
/* bdd_xnor
*/
BDD_STATUS bdd_xnor (Bdd1, Bdd2, BddRes)
    BDD      Bdd1;
    BDD      Bdd2;
    BDD      *BddRes;
{
    return (bdd_equi (Bdd1, Bdd2, BddRes));
}

/*-----
/* bdd_index_occur_rec
*/
-----*/
```

bbdd.c

```
/* CAUTION: The Tag occur bit is used. When we pass a Bdd we set it to 1.*/
/* This means that all the Bdd must have their occur bit set to 0 before*/
/* calling the function. */
*/
static int bdd_index_occur_rec (Bdd, index)
    BDD          Bdd;
    Uint32      index;
{
    BDD_PTR    BddPtr;

    if (ConstantBdd (Bdd))
        return (0);

    BddPtr = GetBddPtrFromBdd (Bdd);

    if (IsBddTagOccurBit (BddPtr)) /* Already traversed ! */
        return (0);

    BddTagOccurBit (BddPtr);

    if (GetBddPtrIndex (BddPtr) == index)
        return (1);

    if (bdd_index_occur_rec (GetBddPtrEdge1 (BddPtr), index))
        return (1);

    if (bdd_index_occur_rec (GetBddPtrEdge0 (BddPtr), index))
        return (1);

    return (0);
}

/*
 * bdd_index_occur
 */
/*
 * Returns 1 if at least one bdd node of index "index" occurs in the BDD/
 * "bdd".
 */
/*
 * PREC: All Occur bit of each Bdd nodes must be set to 0.
 */
int bdd_index_occur (Bdd, index)
    BDD          Bdd;
    Uint32      index;
{
    int       Res;

    Res = bdd_index_occur_rec (Bdd, index);
    (void)ResetTagOccurBit (Bdd);
    return (Res);
}

/*
 */
```

bbdd.c

```
/* FreezeBddNodeRec */  
/*-----*/  
/* Freeze recursively a Bdd node in order to protect it. */  
/*-----*/  
void FreezeBddNodeRec (Bdd) /*-----*/  
    BDD Bdd;  
{  
    BDD_PTR BddPtr;  
  
    if (ConstantBdd (Bdd))  
        return;  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    if (IsBddTagOccurBit (BddPtr)) /* Already traversed ! */  
        return ;  
  
    BddTagOccurBit (BddPtr);  
  
    SetBddPtrRefCount (GetBddPtrFromBdd (Bdd), BDD_MAXREFCOUNT);  
  
    FreezeBddNodeRec (GetBddPtrEdge1 (BddPtr));  
    FreezeBddNodeRec (GetBddPtrEdge0 (BddPtr));  
}  
  
/*-----*/  
/* FreezeBddNode */  
/*-----*/  
/* Freeze the complete Bdd "bdd" by setting its reference to the max */  
/* counter BDD_MAXREFCOUNT. */  
/*-----*/  
/* PREC: the Bdd and its sons must have a Occur bit set to 0. */  
/*-----*/  
void FreezeBddNode (Bdd) /*-----*/  
    BDD Bdd;  
{  
  
    (void) FreezeBddNodeRec (Bdd);  
    (void) ResetTagOccurBit (Bdd);  
}  
  
/*-----*/  
/* pbRec */  
/*-----*/  
static void pbRec (Bdd) /*-----*/  
    BDD Bdd;  
{  
    BDD_PTR BddPtr;  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    if (Bdd == BDD_0)  
    {  
        fprintf (stderr, "(0)");  
    }
```

bbdd.c

```
    return;
}

if (Bdd == BDD_1)
{
    fprintf (stderr, "(1)");
    return;
}

if (Bdd == BDD_Z)
{
    fprintf (stderr, "(Z)");
    return;
}

if (Bdd == BDD_X)
{
    fprintf (stderr, "(X)");
    return;
}

if (BddPtrIsVar (BddPtr))
{
    if (IsBddTypeINV (Bdd))
        fprintf (stderr, "-");

    fprintf (stderr, "%s",
             GetBddVar (GetBddPtrIndex (BddPtr)) ->Name);
    return;
}

if (IsBddTypeINV (Bdd))
    fprintf (stderr, "-");

fprintf (stderr, "[");
fprintf (stderr, "%s",
        GetBddVar (GetBddPtrIndex (BddPtr)) ->Name);
pbRec (GetBddFromBddPtr (GetBddPtrEdge0 (BddPtr)));
fprintf (stderr, " ");
pbRec (GetBddFromBddPtr (GetBddPtrEdge1 (BddPtr)));
fprintf (stderr, "]");

return;
}

/*-----
/* bdd_print
*/
/*-----*/
/* Prints out on the stderr stream the BDD "bdd". */
/*
void bdd_print (Bdd)
    BDD Bdd;
{
    fprintf (stderr, "\n");
    pbRec (Bdd);
    fprintf (stderr, "\n");
}
```

bbdd.c

```
    return;
}

/*
 * bdd_refcount
 */
/*
 * Prints out on the stderr stream the reference counter of Bdd "Bdd".
 */
void bdd_refcount (Bdd)
    BDD Bdd;
{

    fprintf (stderr, "%d\n", GetBddPtrRefCount (GetBddPtrFromBdd (Bdd)));
}

/*
 * bdd_index
 */
/*
 * Prints out on the stderr stream the index of Bdd "Bdd".
 */
void bdd_index (Bdd)
    BDD Bdd;
{

    fprintf (stderr, "%d\n", GetBddPtrIndex (GetBddPtrFromBdd (Bdd)));
}

/*
 * GetBddCache
 */
/*
 * Returns the bdd cache of the current Bdd work space.
 */
BLIST GetBddCache ()
{
    return (GLOB_BDD_WS->BddCache);
}

/*
 * SetWorkSpace
 */
/*
 * Select the current BDD work space. All the BDD operations will be
 * done within this Work Space.
 */
void SetWorkSpace (WorkSpace)
    BDD_WS    WorkSpace;
{
    GLOB_BDD_WS = WorkSpace;
}

/*
 * GetWorkSpace
 */
/*
 * returns the current BDD work space.
 */
```

bbdd.c

```
/*-----*/
BDD_WS GetWorkSpace ()
{
    return (GLOB_BDD_WS);
}

/*-----*/
/* SetNbMaxCreatedNode */
/*-----*/
/* This function allows the user to set the maximum number of nodes that*/
/* can be created physically. This does not correspond finally to the */
/* number of nodes counted at the end of the process (You may create */
/* physically 1000000 nodes for a final Bdd of 1000 nodes). */
/* This is usefull to control the complexity of any Bdd constructions. */
/*-----*/
void SetNbMaxCreatedNode (NbMaxNode)
    int      NbMaxNode;
{
    GLOB_BDD_WS->MaxTotalBddCreated = NbMaxNode;
}

/*-----*/
/* GetNbMaxUsedNode */
/*-----*/
int GetNbMaxUsedNode ()
{
    return (GLOB_BDD_WS->MaxTotalUsedBddNode);
}

/*-----*/
/* GetNbTotalUsedNode */
/*-----*/
int GetNbTotalUsedNode ()
{
    return (GLOB_BDD_WS->TotalUsedBddNode);
}

/*-----*/
/* SetNbMaxUsedNode */
/*-----*/
/* This function allows the user to set the maximum number of nodes that*/
/* can be used in the Bdd cache. This means that it represents the max. */
/* size of the bdd cache allowed. */
/*-----*/
void SetNbMaxUsedNode (NbMaxNode)
    int      NbMaxNode;
{
    GLOB_BDD_WS->MaxTotalUsedBddNode = NbMaxNode;
}

/*-----*/
/* SetUseBddz */
/*-----*/
/*
/* CAUTION ! can slower the BDD package run-time if activated !
*/
/*
/*
/*-----*/
```

bbdd.c

```
/* Ex: Bdd1 = [a [b c (0)] (0)], Bdd2 = [d e (z)] */  
/* if "SetUseBddZ" then we have: */  
/*   (1) Substitute (Bdd1, c, Bdd2) --> [a [d e (X)] (0)] */  
/* */  
/* if not "SetUseBddZ" then we have: */  
/*   (2) Substitute (Bdd1, c, Bdd2) --> [a [d e (Z)] (0)] */  
/* */  
/* Equation (2) is wrong and (1) is right. Unfortunately The option */  
/* makes the BDD package slower. So use it only if you are sure that z */  
/* values are handle in your application. */  
/* */  
/*-----*/  
void SetUseBddZ ()  
{  
    if (GLOB_BDD_WS->BddZ_IsUsed)  
        return;  
  
    fprintf (stderr, "BDD-INFO: Z mode activated\n");  
    GLOB_BDD_WS->BddZ_IsUsed = 1;  
}
```

```
/*-----*/  
/* InitBddWorkSpace */  
/*-----*/  
/* Creates a bdd workspace in which construction of Bdds will be */  
/* performed. The index given for each bdd variable will follow the */  
/* increasing order from [1, 2, ..., x]. The special nodes like BDD_ONE, */  
/* BDD_ZERO have an index of value 'BDD_MAXINDEX'. A bdd work space */  
/* 'WorkSpace' is returned if everything is OK. */  
/*-----*/  
BDD_STATUS InitBddWorkSpace (MaxVar, WorkSpace)  
{  
    int             MaxVar;  
    BDD_WS         *WorkSpace;  
  
    OBJECT_PAGE    NewPage;  
    BLIST          NewBddCache;  
    APPLY_CACHE    NewApplyCache;  
    BDD_PTR        NewBdd;  
    BDD_STATUS     Status;  
    BLIST          NewList;  
  
    if ((*WorkSpace = (BDD_WS)malloc (1, sizeof(BDD_WS_REC))) == NULL)  
        return (BDD_MEMORY_FULL);  
  
    (*WorkSpace)->TotalMemoryAlloc = 1*sizeof(BDD_WS_REC);  
  
    SetWorkSpace (*WorkSpace);  
  
    if (BListCreateWithSize (MaxVar+1, &NewList))  
        return (BDD_MEMORY_FULL);  
    (*WorkSpace)->Var = NewList;  
  
    /* Allocating first page of BDD_REC objects. */  
    if ((Status = PageAllocate (BDD_PAGE_SIZE, sizeof(BDD_REC), &NewPage)))  
        return (Status);
```

bbdd.c

```
(*WorkSpace) ->CurrentBddPage = NewPage;

/* Allocating first page of APPLY_CELL_REC objects. */
if ((Status = PageAllocate (APPLY_PAGE_SIZE, sizeof(APPLY_CELL_REC),
                           &NewPage)))
    return (Status);
(*WorkSpace) ->CurrentApplyCellPage = NewPage;

/* Allocating related apply caches */
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->OrCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->AndCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->ImpCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->EqCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->RstCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->JoinCache = NewApplyCache;

(*WorkSpace) ->MaxTotalBddCreated = DEFAULT_MAX_NODE;
(*WorkSpace) ->MaxTotalUsedBddNode = DEFAULT_MAX_NODE;

/* Allocating the BDD cache. */
if ((Status = CreateBddCache (MaxVar, &NewBddCache)))
    return (Status);
(*WorkSpace) ->BddCache = NewBddCache;

/* Increasing order. */
(*WorkSpace) ->IndexVarOrder = 0;
(*WorkSpace) ->CompareIdx = CompareIndex;

/* Create the 1 BDD node which corresponds to the logical 1. */
/* Its index is the highest one so it is always pushed to the leaves */
/* during the Apply calls. */
if ((Status = CreateBddNode (&NewBdd)))
    return (Status);

SetBddPtrIndex (NewBdd, BDD_MAXINDEX);

SetBddTypeDIR ((*WorkSpace) ->BddOne, NewBdd);
SetBddTypeINV ((*WorkSpace) ->BddZero, NewBdd);

/* Creates the Z BDD node. */
SetBddPtrZ ((*WorkSpace) ->BddZ, NewBdd);

/* Creates the X BDD node. */

```

```

SetBddPtrX ((*WorkSpace) ->BddX, NewBdd);

/* Initialize field 'UniqueId' to 0. */  

(*WorkSpace) ->UniqueId = 0;

/* adding a fake variable as first variable. */  

/* The very first variable will start at Index = 1 in the list and */  

/* will have the bdd index = 1. */  

if (BListAddElt (GLOB_BDD_WS->Var, NULL))  

    return (BDD_MEMORY_FULL);

/* We add also a fake Bdd cache as first cache */  

if (BListAddElt (GLOB_BDD_WS->BddCache, NULL))  

    return (BDD_MEMORY_FULL);

return (BDD_OK);
}

/*-----*/  

/* InitBddWorkSpaceInvOrder */  

/*-----*/  

/* Creates a bdd workspace in which construction of Bdds will be */  

/* performed. The index given for each bdd variable will follow the */  

/* decreasing order from [x, ..., 2, 1, 0]. The special node like */  

/* BDD_ONE, BDD_ZERO have an index of -1. */  

/* A bdd work space 'WorkSpace' is returned if everything is OK. */  

/*-----*/  

BDD_STATUS InitBddWorkSpaceInvOrder (MaxVar, WorkSpace)
{
    int          MaxVar;
    BDD_WS      *WorkSpace;
    OBJECT_PAGE  NewPage;
    BLIST        NewBddCache;
    APPLY_CACHE   NewApplyCache;
    BDD_PTR      NewBdd;
    BDD_STATUS    Status;
    BLIST        NewList;

    if ((*WorkSpace = (BDD_WS)malloc (1, sizeof(BDD_WS_REC))) == NULL)
        return (BDD_MEMORY_FULL);

    (*WorkSpace) ->TotalMemoryAlloc = 1*sizeof(BDD_WS_REC);

    SetWorkSpace (*WorkSpace);

    if (BListCreateWithSize (MaxVar+1, &NewList))
        return (BDD_MEMORY_FULL);
    (*WorkSpace) ->Var = NewList;

    /* Allocating first page of BDD_REC objects. */  

    if ((Status = PageAllocate (BDD_PAGE_SIZE, sizeof(BDD_REC), &NewPage)))
        return (Status);
    (*WorkSpace) ->CurrentBddPage = NewPage;

    /* Allocating first page of APPLY_CELL_REC objects. */  


```

```

if ((Status = PageAllocate (APPLY_PAGE_SIZE, sizeof(APPLY_CELL_REC),
                           &NewPage)))
    return (Status);
(*WorkSpace) ->CurrentApplyCellPage = NewPage;

/* Allocating related apply caches */ 
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->OrCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->AndCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->ImpCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->EqCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->RstCache = NewApplyCache;
if ((Status = CreateApplyCache (&NewApplyCache)))
    return (Status);
(*WorkSpace) ->JoinCache = NewApplyCache;

(*WorkSpace) ->MaxTotalBddCreated = DEFAULT_MAX_NODE;
(*WorkSpace) ->MaxTotalUsedBddNode = DEFAULT_MAX_NODE;

/* Allocating the BDD cache. */ 
if ((Status = CreateBddCache (MaxVar, &NewBddCache)))
    return (Status);
(*WorkSpace) ->BddCache = NewBddCache;

/* Decreasing order. */ 
(*WorkSpace) ->IndexVarOrder = 1;
(*WorkSpace) ->CompareIdx = CompareIndex_ec;

/* Create the 1 BDD node which corresponds to the logical 1. */ 
/* Its index is the highest one so it is always pushed to the leaves */ 
/* during the Apply calls. */ 
if ((Status = CreateBddNode (&NewBdd)))
    return (Status);

SetBddPtrIndex (NewBdd, -1);

SetBddTypeDIR ((*WorkSpace) ->BddOne, NewBdd);
SetBddTypeINV ((*WorkSpace) ->BddZero, NewBdd);

/* Creates the Z BDD node. */ 
SetBddPtrZ ((*WorkSpace) ->BddZ, NewBdd);

/* Creates the X BDD node. */ 
SetBddPtrX ((*WorkSpace) ->BddX, NewBdd);

/* Initialize field 'UniqueId' to -1. */ 
(*WorkSpace) ->UniqueId = -1;

```

```
    return (BDD_OK);  
}  
  
/*-----*/
```

bbdd.e

```
/*-----*/
/*
/*      File:          bbdd.e
/*      Version:        1.1
/*      Modifications: -
/*      Documentation: -
/*
/*      Class: none
/*      Inheritance:
/*
/*-----*/
/*-----*/
/* GLOBAL VARIABLE */
/*-----*/
extern BDD_WS      GLOB_BDD_WS;      /* Default BDD Work Space */

/*-----*/
/* InitBddWorkSpace */
/*-----*/
extern BDD_STATUS InitBddWorkSpace /* int MaxVar, BDD_WS *WorkSpace */;

/*-----*/
/* SetUseBddZ */
/*-----*/
/*
/* CAUTION ! can slower the BDD package run-time if activated !
/*
/* Ex: Bdd1 = [a [b c (0) (0)], Bdd2 = [d e (Z)]           */
/* if "SetUseBddZ" then we have:                           */
/*   (1) Substitute (Bdd1, c, Bdd2) --> [a [d e (X)] (0)]   */
/* if not "SetUseBddZ" then we have:                         */
/*   (2) Substitute (Bdd1, c, Bdd2) --> [a [d e (Z)] (0)]   */
/* Equation (2) is wrong and (1) is right. Unfortunately The option */
/* makes the BDD package slower. So use it only if you are sure that Z */
/* values are handle in your application.                   */
/*
/*-----*/
extern void SetUseBddZ ();

/*-----*/
/* SetWorkSpace */
/*-----*/
/*
/* Select the current BDD work space. All the BDD operations will be */
/* done within this Work Space.                                     */
/*
/*-----*/
extern void SetWorkSpace ();

/*-----*/
```

bbdd.e

```
/* GetWorkSpace */  
/*-----*/  
/* returns the current BDD work space. */  
/*-----*/  
extern BDD_WS GetWorkSpace ();  
  
/*-----*/  
/* GetBddCache */  
/*-----*/  
/* Returns the Bdd cache of the current Bdd work space. */  
/*-----*/  
extern BDD_CACHE GetBddCache ();  
  
/*-----*/  
/* bdd_print */  
/*-----*/  
/* Prints out on the stderr stream the BDD "bdd". */  
/*-----*/  
extern void bdd_print /* BDD Bdd */;  
  
/*-----*/  
/* bdd_refcount */  
/*-----*/  
/* Prints out on the stderr stream the reference counter of Bdd "Bdd". */  
/*-----*/  
extern void bdd_refcount /* BDD Bdd */;  
  
/*-----*/  
/* bdd_index */  
/*-----*/  
/* Prints out on the stderr stream the index of Bdd "Bdd". */  
/*-----*/  
extern void bdd_index /* BDD Bdd */;  
  
/*-----*/  
/* bdd_freeze */  
/*-----*/  
/* Set the reference counter of bdd node "bdd" to BDD_MAXREFCOUNT in */  
/* order to freeze it. */  
/*-----*/  
extern void bdd_freeze /* BDD Bdd */;  
  
/*-----*/  
/* UseBdd */  
/*-----*/  
/* Increments the reference counter of the bdd node "Bdd" and returns */  
/* its argument. */  
/*-----*/  
extern BDD UseBdd /* BDD Bdd */;
```

bbdd.e

```
/*-----*/
/* FreezeBddNode */
/*-----*/
/* Freeze the complete Bdd "bdd" by setting its reference to the max      */
/* counter BDD_MAXREFCOUNT.                                              */
/*-----*/
/* PREC: the Bdd and its sons must have a Occur bit set to 0.           */
/*-----*/
extern void FreezeBddNode /* BDD Bdd */;

/*-----*/
/* ResetTagOccurBitInWs */
/*-----*/
/* Reset all the Occur bits of all the bdd nodes in the work space "WS". */
/* This is a brutal but safe reset.                                         */
/*-----*/
extern void ResetTagOccurBitInWs /* BDD_WS Ws */;

/*-----*/
/* SquizWorkSpace */
/*-----*/
/* This function removes from the default Bdd work space all Bdd nodes   */
/* which are not frozen (e.g with reference count < BDD_MAXREFCOUNT. It */
/* reste also completely all the Apply cache tables.                      */
/*-----*/
extern void SquizDefaultWorkSpace ();

/*-----*/
/* bdd_index_occur */
/*-----*/
/* Returns 1 if at least one bdd node of index "index" occurs in the BDD*/
/* "bdd" and 0 otherwise.                                                 */
/*-----*/
/* PREC: All Occur bit of each Bdd nodes must be set to 0.             */
/*-----*/
extern int bdd_index_occur /* BDD Bdd, Uint32 index */;

/*-----*/
/* SetNbMaxUserNode */
/*-----*/
/* This function allows the user to set the maximum number of nodes that */
/* can be used in the Bdd cache. This means that it represents the max. */
/* size of the bdd cache allowed.                                         */
/*-----*/
extern void SetNbMaxUserNode ();

/*-----*/
/* GetNbTotalUsedNode */
/*-----*/
extern int GetNbTotalUsedNode ();
```

bbdd.e

```
/*-----*/
/* GetNbMaxUsedNode */
/*-----*/
extern int GetNbMaxUsedNode ();
```



```
/*-----*/
/* SetNbMaxCreatedNode */
/*-----*/
/* This function allows the user to set the maximum number of nodes that*/
/* can be created physically. This does not correspond finally to the */
/* number of nodes counted at the end of the process (You may create */
/* physically 1000000 nodes for a final Bdd of 1000 nodes). */
/* This is usefull to control the complexity of any Bdd constructions. */
/*-----*/
extern void SetNbMaxCreatedNode /* int NbMaxNode */;
```



```
/*-----*/
/* BddCountNodesAndMark */
/*-----*/
/* CAUTION!: two LSB bits of Field Hook can be modified so take care if */
/* you use this field ! */
/* The nodes are marked thru the Field Hook so if you rerun the same */
/* function it will not count the previous visited nodes. */
/* This function updates also Field ->TotalCountBddNode of the current */
/* BDD work space. */
/*-----*/
extern BDD_STATUS BddCountNodesAndMark /* BDD Bdd, int *NbNode */;
```



```
/*-----*/
/* BddCountNodes */
/*-----*/
/* Counts the exact number of nodes of the bdd "Bdd" but removes marks */
/* after traversal in order to do not have side effects on next call to */
/* this function. */
/*-----*/
extern void BddCountNodes /* BDD Bdd, int *NbNode */;
```



```
/*-----*/
/* BddPtrIsVar */
/*-----*/
extern int BddPtrIsVar /* BDD_PTR BddPtr */;
```



```
/*-----*/
/* bdd_one */
/*-----*/
/* Returns the Bdd (1) of the current bdd work space. */
/*-----*/
extern BDD bdd_one ();
```



```
/*-----*/
/* bdd_zero */
/*-----*/
```

bbdd.e

```
/*-----*/
/* Returns the Bdd (0) of the current bdd work space.          */
/*-----*/
extern BDD bdd_zero ();

/*-----*/
/* bdd_Z                                         */
/*-----*/
/* Returns the Bdd (Z) of the current bdd work space.          */
/*-----*/
extern BDD bdd_Z ();

/*-----*/
/* bdd_X                                         */
/*-----*/
/* Returns the Bdd (X) of the current bdd work space.          */
/*-----*/
extern BDD bdd_X ();

/*-----*/
/* bdd_not                                       */
/*-----*/
/* Boolean NOT function applied.                      */
/*-----*/
extern BDD_STATUS bdd_not /* BDD Bdd1, BDD *BddRes */;

/*-----*/
/* bdd_or                                         */
/*-----*/
/* Boolean OR function applied.                     */
/*-----*/
extern BDD_STATUS bdd_or /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;

/*-----*/
/* bdd_nor                                       */
/*-----*/
/* Boolean NOR function applied.                   */
/*-----*/
extern BDD_STATUS bdd_nor /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;

/*-----*/
/* bdd_and                                       */
/*-----*/
/* Boolean AND function applied.                  */
/*-----*/
extern BDD_STATUS bdd_and /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;

/*-----*/
/* bdd_nand                                      */
/*-----*/
/*-----*/
```

bbdd.e

```
/* Boolean NAND function applied. */  
/*-----*/  
extern BDD_STATUS bdd_nand /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_xor */  
/*-----*/  
/* Boolean XOR function applied. */  
/*-----*/  
extern BDD_STATUS bdd_xor /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_xnor */  
/*-----*/  
/* Boolean XNOR function applied. */  
/*-----*/  
extern BDD_STATUS bdd_xnor /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_impl */  
/*-----*/  
/* Boolean IMPLICATION function applied. */  
/*-----*/  
extern BDD_STATUS bdd_impl /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_equi */  
/*-----*/  
/* Boolean EQUIVALENCE function applied. */  
/*-----*/  
extern BDD_STATUS bdd_equi /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_restrict */  
/*-----*/  
/* Boolean RESTRICT function applied. */  
/*-----*/  
extern BDD_STATUS bdd_restrict /* BDD Bdd1, BDD dmnn, BDD *BddRes */;  
  
/*-----*/  
/* bdd_join */  
/*-----*/  
/* Boolean JOIN function applied. Represents the resolution function ( */  
/* defined by VHDL IEEE std_logic_1164). */  
/*-----*/  
extern BDD_STATUS bdd_join /* BDD Bdd1, BDD Bdd2, BDD *BddRes */;  
  
/*-----*/  
/* bdd_compose */  
/*-----*/
```

bbdd.e

```
/* Substitutes variable of index "index" with bdd "BddSub" in the bdd      */
/* "Bdd".                                         */
/*                                                 */
/* PREC!: The tag occur bit must be set to 0 for all the bdd nodes of      */
/* "Bdd".                                         */
/*-----*/
extern BDD_STATUS bdd_compose /* Bdd, index, BddSub, BddRes */;

/*-----*/
/* bdd_cofac0                                     */
/*-----*/
/* Builds the cofactor 0 of variable index "index" of Bdd "Bdd". */
/*-----*/
extern BDD_STATUS bdd_cofac0 /* Bdd, index, BddRes */;

/*-----*/
/* bdd_cofac1                                     */
/*-----*/
/* Builds the cofactor 1 of variable index "index" of Bdd "Bdd". */
/*-----*/
extern BDD_STATUS bdd_cofac1 /* Bdd, index, BddRes */;
```

bbdd.h

```
/*
 *-----*/
/*
 *      File:          bbdd.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*
 *-----*/
#ifndef BBDD_H
#define BBDD_H

#include <stdio.h>

#define GetBddVar(index)      ((BDD_VAR)BListElt (GLOB_BDD_WS->Var, index))
#define GetWsBddVar(Ws, index) ((BDD_VAR)BListElt (Ws->Var, index))
#define GetCache(cache, index) ((BDD_CACHE)BListElt (cache, index))

/*=====
/* CRITICAL PARAMETERS
 */
/*=====

/* Allocation of pages of BDDs objects
#define BDD_PAGE_SIZE          4096

/* Allocation of pages for applys objects
#define APPLY_PAGE_SIZE         4096

/* Parameters for Operator cache tables. Tables are of size :
/* APPLY_CACHE_SIZE . MAX_NB_APPLY_CELLS . sizeof (APPLY_CELL_REC)
#define MAX_NB_APPLY_CELLS      10

#define APPLY_CACHE_SIZE        10000
#define MAX_APPLY_CACHE_SIZE    200000

/* Parameter for BDD cache table.
#define BDD_PTR_SUB_CACHE_SIZE      500
#define MAX_BDD_CACHE_SIZE        600000

/* Parameter to control rehashing of caches.
#define MAX_APPLY_CONFLICT        100000
#define MAX_BDD_CONFLICT          100000

/* Parameter to control complexity during BDD construction
#define DEFAULT_MAX_NODE          5000000

/*=====*/
```

```

/*
 *-----*
 /* BASIC types                               */
 *-----*/
typedef unsigned int     Uint32;      /* 32 bit field           */
                                         /* */
typedef Uint32          BDD;         /* Typed BDD              */
                                         /* */

/*
 *-----*
 /* Generic PAGE type to store different kinds of objects */
 *-----*/
typedef struct OBJECTPAGEREC {
    int             NbObj;        /* # of objects in the page */
    int             FirstObject;
    int             NextFreeObject;
    int             LastObject;
    struct OBJECTPAGEREC *NextPage;
    int             PageFull;     /* 1 --> page full, 0 otherwise */
} OBJECT_PAGE_REC;

typedef OBJECT_PAGE_REC *OBJECT_PAGE;

/*
 *-----*
 /* Basic structure of a BDD node.           */
 /* Layout of BDD_REC record:                */
 /*-----*/
/* +-----+
/* | 31.....12 | 11 | 10 | 9 ..... 0 |
/* +-----+-----+-----+-----+-----+
/* |       Index      | Tag1 | Tag0 |       RefCount   |
/* +-----+-----+-----+-----+-----+
/* |                           Edge[0]           |
/* +-----+-----+
/* |                           Edge[1]           |
/* +-----+-----+
/* |                           Cache Compose |
/* +-----+-----+
/* |                           Next            |
/* +-----+-----+
/* |                           Hook            |
/* +-----+-----+
/* +-----+-----+
/* variable id:      is on (31-12+1) = 20 bits so we can accept up to*/
/* 2^20 = 1048576 variables.                                */
/* refcount:      is on (9-0+1) = 10 bits = 1024               */
/* Tag1:          is a bit Tag used for traversal algorithms (ex  */
/* BDD count node computation)                            */
/* Tag0:          is a bit Tag used for traversal algorithms (ex  */
/* var occur computation)                                */
/* Sizeof (BDD_REC) = 28 Bytes.                         */
/* 1 Million BDD nodes = 28 MegaB.                      */
/*-----*/

```

bbdd.h

```
typedef struct BDD_STRUCT {
    Uint32             Header;           /* cf. encoding above          */
    BDD               Edge[2];          /* * 0 and 1 edge sons          */
    BDD               CacheCompose;
    struct BDD_STRUCT *Next;            /* next BDD in the Bdd cache */
    Uint32             Hook;             /* for user purpose           */
} BDD_REC;

#define IndexMask 0xFFFFF000 /* Index Mask                  */
#define RefCountMask 0x000003FF /* RefCount Mask               */
#define IndexMaskC 0x00000FFF /* Complement of Index Mask   */
#define RefCountMaskC 0xFFFFFC00 /* Complement of RefCount Mask */
#define CountBitMask 0x00000800 /* Count bit mask              */
#define CountBitMaskC 0xFFFFF7FF /* Complement of Count bit mask */
#define OccurBitMask 0x00000400 /* Occur bit mask              */
#define OccurBitMaskC 0xFFFFFBFF /* Complement of Occur bit mask */

/*-----*/
/* index operators                         */
/*-----*/
#define GetBddPtrIndex(BddPtr)      ((BddPtr)->Header >> 13)
#define SetBddPtrIndex(BddPtr, Index)\
    ((BddPtr)->Header = (((BddPtr)->Header & IndexMaskC) | (Index << 13)))

#define BDD_MAXINDEX ((Uint32)0x000FFFFF)

/*-----*/
/* Ref Count operators                     */
/*-----*/
#define GetBddPtrRefCount(BddPtr)   (((BddPtr)->Header) & RefCountMask)
#define SetBddPtrRefCount(BddPtr, RefCount)\
    ((BddPtr)->Header = (((BddPtr)->Header & RefCountMaskC) | RefCount))

#define BDD_MAXREFCOUNT RefCountMask

/*-----*/
/* Edges operators                        */
/*-----*/
#define GetBddPtrEdge1(BddPtr)      ((BddPtr)->Edge[1])
#define SetBddPtrEdge1(BddPtr, T)   ((BddPtr)->Edge[1] = T)

#define GetBddPtrEdge0(BddPtr)      ((BddPtr)->Edge[0])
#define SetBddPtrEdge0(BddPtr, E)   ((BddPtr)->Edge[0] = E)

/*-----*/
/* Next bdd operators                    */
/*-----*/
#define GetBddPtrNext(BddPtr)      ((BddPtr)->Next)
#define SetBddPtrNext(BddPtr, N)   ((BddPtr)->Next = N)

/*-----*/
/* count bit operators                  */
/*-----*/
```

bbdd.h

bbdd.h

```
#define BDD_CONST_MASK_C      0xFFFFFFFFC /* Mask to remove constant */
                                         /* encoded bits.           */

#define GetBddType(Bdd)          ((Bdd) & BDDTypeMask)
#define GetBddPtr(Bdd)           ((BDD_PTR)((Bdd) & BDD_PTRMask))

#define SetBddTypeDIR(Bdd1, Bdd2) (Bdd1 = (BDD)((int)Bdd2) &
                                BDDTypeMaskC)
#define SetBddTypeINV(Bdd1, Bdd2) (Bdd1 = (BDD)((int)Bdd2) | BDDTypeMask)
#define SetBddPtrZ(Bdd1, Bdd2)   (Bdd1 = (BDD)((int)Bdd2) | BDD_Z_MASK)
#define SetBddPtrX(Bdd1, Bdd2)   (Bdd1 = (BDD)((int)Bdd2) | BDD_X_MASK)

#define IsBddTypeDIR(Bdd)        (!GetBddType(Bdd))
#define IsBddTypeINV(Bdd)        (GetBddType(Bdd))

#define BddRmConstant(Bdd)       (Bdd & BDD_CONST_MASK_C)
#define BddConstantIndex(Bdd)   (Bdd & BDD_CONST_MASK)
                           /* 4 values:                */
                           /* 0 -> BDD_ONE            */
                           /* 1 -> BDD_ZERO            */
                           /* 2 -> BDD_Z               */
                           /* 3 -> BDD_X               */

#define GetBddFromBddPtr(BddPtr) ((BDD)BddPtr)

/*-----
/* ConstantBdd
/*-----
/* Constants are 1, 0, X, Z.
#define ConstantBdd(Bdd) ((BddRmConstant(Bdd) == BDD_1) ? 1 : 0)

#define BDD_1              (GLOB_BDD_WS->BddOne)
#define BDD_0              (GLOB_BDD_WS->BddZero)
#define BDD_Z              (GLOB_BDD_WS->BddZ)
#define BDD_X              (GLOB_BDD_WS->BddX)

#define BDD_1_WS(ws)        (ws->BddOne)
#define BDD_0_WS(ws)        (ws->BddZero)
#define BDD_Z_WS(ws)        (ws->BddZ)
#define BDD_X_WS(ws)        (ws->BddX)

#define BDD_Z_NOT_USED    (GLOB_BDD_WS->BddZ_IsUsed == 0)

/*-----
/* ComplementBDD
/*-----
/* We complement only the pointer only if it not the BDD_Z and BDD_X
/*-----
/* THIS MACRO IS NOT SUPPORTED WITH THE GCC
#define ComplementBDD(x) (((x == BDD_X) || (x == BDD_Z)) ? BDD_X : (x ^
BDDTypeMask))
*/
```

```

/*-----*/
/* Type APPLY_CACHE */
/*-----*/
typedef struct APPLY_CELL_STRUCT {
    BDD_PTR             Bdd1;
    BDD_PTR             Bdd2;
    BDD                 BddRes;
    struct APPLY_CELL_STRUCT *Next;
} APPLY_CELL_REC;

typedef APPLY_CELL_REC *APPLY_CELL;

typedef struct APPLY_CACHE_STRUCT {
    int                 NbConflicts;
    int                 CacheSize;
    APPLY_CELL *Buckets;           /* Of size CacheSize */
} APPLY_CACHE_REC;

typedef APPLY_CACHE_REC *APPLY_CACHE;

/*-----*/
/* Type BDD_CACHE */
/*-----*/
typedef struct BDD_CACHE_STRUCT {
    int                 CacheSize;
    BDD_PTR             *Buckets;           /* Of size CacheSize */
    int                 NbConflicts;
    int                 NbDeadBddNode;
    int                 NbBddNode;
} BDD_CACHE_REC;

typedef BDD_CACHE_REC *BDD_CACHE;

/*-----*/
/* Type BDD_VAR */
/*-----*/
typedef enum { BDD_INPUT, BDD_OUTPUT, BDD_INOUT } DIRECTION;

typedef struct BDD_VAR_STRUCT {
    char                *Name;
    BDD                 Bdd;
    int                 Id;
    struct BDD_VAR_STRUCT *Next;
    int                 *Hook;
} BDD_VAR_REC;

typedef BDD_VAR_REC *BDD_VAR;

#define BddVarName(b)          (((b)->Name))
#define BddVarBdd(b)           (((b)->Bdd))
#define BddVarId(b)            (((b)->Id))
#define BddVarNext(b)          (((b)->Next))
#define BddVarHook(b)          (((b)->Hook))

```

```

#define SetBddVarName(b,n)          (((b)->Name = n))
#define SetBddVarBdd(b,n)           (((b)->Bdd = n))
#define SetBddVarId(b,n)            (((b)->Id = n))
#define SetBddVarNext(b,n)          (((b)->Next = n))
#define SetBddVarHook(b,n)          (((b)->Hook = n))

/*-----
/* Environment structure type to manage any BDDs
/*-----*/
typedef int (*CmpFunc) (Uint32 Idx1, Uint32 Idx2);

typedef struct BDD_WS_STRUCT {

    /* General informations about the BDD work space */
    char      *WorkSpaceName;        /* name of the work space */
    BLIST     Var;                 /* List of BDD_VAR */
    int       UniqueId;           /* next free unique id. */
    /* Represents the number of Var */
    /* created in The Bdd Ws. */
    int       IndexVarOrder;        /* 0/1 -> Variable are sorted in*/
    /* increasing/decreasing index */
    /* order. */
    /* Default is increasing order */
    CmpFunc   CompareIdx;

    /* Memory Manager related informations */
    OBJECT_PAGE CurrentApplyCellPage; /* Page of Apply cell rec.*/
    APPLY_CELL FreedApplyCell;

    OBJECT_PAGE CurrentBddPage;        /* Page of BDD records */
    BDD_PTR     FreedBdd;             /* List of freed BDD_PTR */

    /* All the Caches (Hash tables) */
    APPLY_CACHE OrCache;
    APPLY_CACHE AndCache;
    APPLY_CACHE ImpCache;
    APPLY_CACHE EqCache;
    APPLY_CACHE RstCache;
    APPLY_CACHE JoinCache;

    BLIST     BddCache;              /* List of BDD_CACHE */

    /* Specific BDD constants */
    BDD      BddOne;                /* The constant BDD 1. */
    BDD      BddZero;                /* The constant BDD 0. */
    BDD      BddZ;                  /* High impedance. */
    BDD      BddX;                  /* Conflict. */

    int      BddZ_IsUsed;           /* 1 --> Z is used in the
                                    /* package, 0 -> not used */

    /* BDD related informations. */
    int      MaxTotalBddCreated;    /* Max physical Bdd nodes */
}

```

```

        /* allowed.          */

int      MaxTotalUsedBddNode;    /* Max used Bdd nodes in   */
                                /* the Bdd cache.         */

int      TotalBddCreated;       /* Number of physical      */
                                /* created Bdd nodes.     */

int      TotalUsedBddNode;      /* Number of current nodes*/
                                /* in the Bdd cache.     */

int      TotalCountBddNode;     /* Number of current nodes*/
                                /* counted and Marked. */

int      TotalBddDead;          /* Number of dead nodes */

int      TotalBddConflicts;     /* Number of conflict when*/
                                /* searching matching in   */
                                /* the Bdd cache.         */

/* APPLY related informations.          */
int      TotalApplyConflicts;
int      TotalApplySuccess;
int      TotalApplyCreated;

int      TotalMemoryCalloc;      /* Total memory allocated */
                                /* in this BDD work space */

} BDD_WS_REC;

typedef BDD_WS_REC *BDD_WS;

/*-----
/* Type BDD_STATUS
/*-----*/
/*-----*/
/* ERROR returned codes during BDD construction and manipulation. */
/*-----*/
/*-----*/
typedef enum {

        BDD_OK,                  /* Everything OK !           */

        BDD_MAX_NODE_CREATED,     /* Error if the number of physical */
                                /* nodes created is greater than a   */
                                /* certain number specified by the user */

        BDD_MAX_NODE_USED,        /* Error if the number of used       */
                                /* bdd nodes is greater than a       */
                                /* certain number specified by the user */

        BDD_MAX_VAR_ID_REACHED,   /* Max. Id Variables reached.      */

        BDD_CACHE_TOO_SMALL,      /* Specified a too small BDD cache */
                                /* compare to # variable created after */
                                /* */

```

bdd.h

```
BDD_MEMORY_FULL      /* No more memory available          */  
} BDD_STATUS;  
  
/*----- EOF -----*/  
#endif
```

bbddmem.c

```
/*
 *-----*/
/* File: bbddmem.c
 * Version: 1.1
 * Modifications: -
 * Documentation: -
 */
/* Class: none
 * Inheritance:
 */
/*-----*/
#include <stdio.h>

#ifndef MEMORY /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "bbdd.h"

/*
 *-----*/
/* EXTERN */
/*-----*/
extern BDD_WS GLOB_BDD_WS; /* Current global BDD work space */
extern BDD_PTR GetBddPtrFromBdd ();
extern Uint32 GetBddPtrRefCount ();

/*
 *-----*/
/* BDD_CALLOC */
/*-----*/
/* Basic allocation in the BDD work space. If STAT is define then we
 * compute the memory allocated and store this information in the BDD
 * work space. */
/*-----*/
#ifdef STAT
Uint32 BDD_CALLOC (NbObj, SizeObj)
    Uint32 NbObj;
    Uint32 SizeObj;
{
    GLOB_BDD_WS->TotalMemoryAlloc += NbObj*SizeObj;
    return ((Uint32)malloc (NbObj, SizeObj));
}
#else
#define BDD_CALLOC malloc
#endif

/*
 *-----*/
/* PageAllocate */
/*-----*/
/* Allocate a memory page of 'NbObj' objects of size 'SizeObj'. */
/*-----*/
/* ERROR: if no more memory, BDD_MEMORY_FULL is returned. */
/*-----*/
BDD_STATUS PageAllocate (NbObj, SizeObj, NewPage)
    int NbObj;
```

bbddmem.c

```
int           SizeObj;
OBJECT_PAGE   *NewPage;
{

    if ((*NewPage = (OBJECT_PAGE)
          BDD_CALLOC (1, sizeof(OBJECT_PAGE_REC))) == NULL)
    {
        return (BDD_MEMORY_FULL);
    }

    if (((*NewPage)->FirstObject = (int)BDD_MALLOC (1, NbObj*SizeObj)) ==
        (int)NULL)
    {
        free ((char*)*NewPage);
        *NewPage = NULL;
        return (BDD_MEMORY_FULL);
    }

    (*NewPage)->NbObj = NbObj;
    (*NewPage)->NextFreeObject = (*NewPage)->FirstObject;
    (*NewPage)->LastObject = (*NewPage)->FirstObject + SizeObj*(NbObj-1);
    (*NewPage)->NextPage = NULL;

    return (BDD_OK);
}

/*-----
/* GetFreedBdd
*/
/*-----*/
/* Returns the first Bdd node previously freed and returns it. All the */
/* fields of the returned BDD_PTR are set to NULL. If not available */
/* BDD_PTR encountered then BDD_PTR_NULL is returned. */
/*
/* ERROR: none
*/
/*-----*/
BDD_PTR GetFreedBdd ()
{
    BDD_PTR       FreedBdd;

    /* actually no freed Bdd yet ... */
    if (GLOB_BDD_WS->FreedBdd == NULL)
        return (BDD_PTR_NULL);

    FreedBdd = GLOB_BDD_WS->FreedBdd;
    GLOB_BDD_WS->FreedBdd = FreedBdd->Next;

    memset((char*)FreedBdd, 0, sizeof(BDD_REC));

    return (FreedBdd);
}

/*-----
/* GetFreedApplyCell
*/
/*-----*/
```

bbddmem.c

```
/* Returns the first APPLY_CELL previously freed and returns it. All the*/
/* fields of the returned APPLY_CELL are set to NULL. If not available */
/* APPLY_CELL then return NULL. */
*/
/* ERROR: none */
*-----
APPLY_CELL GetFreedApplyCell ()
{
    APPLY_CELL            FreedApplyCell;

    /* actually no freed APPLY_CELL yet ... */
    if (GLOB_BDD_WS->FreedApplyCell == NULL)
        return (NULL);

    FreedApplyCell = GLOB_BDD_WS->FreedApplyCell;
    GLOB_BDD_WS->FreedApplyCell = FreedApplyCell->Next;

    memset((char*)FreedApplyCell, 0, sizeof(APPLY_CELL_REC));

    return (FreedApplyCell);
}

*-----
/* CreateBddNode */
*-----
/* Creation of a basic Bdd node either coming from the previous freed */
/* Bdds or from a real physical allocation. */
*-----
BDD_STATUS CreateBddNode (Bdd)
    BDD_PTR *Bdd;
{
    OBJECT_PAGE    CurrentBddPage;
    OBJECT_PAGE    NewPage;
    BDD_STATUS     Status;

    /* Another Bdd node used. */
    GLOB_BDD_WS->TotalUsedBddNode++;

    /* Too many used Bdd nodes in the Bdd cache. */
    if (GLOB_BDD_WS->TotalUsedBddNode > GLOB_BDD_WS->MaxTotalUsedBddNode)
        return (BDD_MAX_NODE_USED);

    /* First we try to pick up a previous freed BDD */
    if ((*Bdd = GetFreedBdd ()) != BDD_NULL)
        return (BDD_OK);

    /* If not, we try to pick up one in the current allocated BDD page */
    CurrentBddPage = GLOB_BDD_WS->CurrentBddPage;

    /* If the current page is full then we create a new one. */
    if (CurrentBddPage->PageFull)
    {
```

bbddmem.c

```
if ((Status = PageAllocate (BDD_PAGE_SIZE, sizeof(BDD_REC),
&NewPage)) )
    return (Status);

    NewPage->NextPage = CurrentBddPage;
    GLOB_BDD_WS->CurrentBddPage = NewPage;
    CurrentBddPage = GLOB_BDD_WS->CurrentBddPage;
}

*Bdd = (BDD_PTR)CurrentBddPage->NextFreeObject;
if (CurrentBddPage->NextFreeObject == CurrentBddPage->LastObject)
    CurrentBddPage->PageFull = 1;
else
    CurrentBddPage->NextFreeObject += sizeof(BDD_REC);

/* The Bdd node is created physically */  

GLOB_BDD_WS->TotalBddCreated++;

/* Too many physical Bdd nodes created. */  

if (GLOB_BDD_WS->TotalBddCreated > GLOB_BDD_WS->MaxTotalBddCreated)
    return (BDD_MAX_NODE_CREATED);

return (BDD_OK);
}

/*-----*/  

/* FrozenBddNode */  

/*-----*/  

/* We froze the Bdd node by setting its ref count to the max one. This */  

/* protect the Bdd from any Free afterwhile. */  

/*-----*/  

int FrozenBddNode (BddPtr)
    BDD_PTR BddPtr;
{
    return (GetBddPtrRefCount (BddPtr) == BDD_MAXREFCOUNT);
}

/*-----*/  

/* FreePhysicalBdd */  

/*-----*/  

/* We add the Bdd Ptr "BddPtr" in the global list of previously freed */  

/* Bdd nodes "GLOB_BDD_WS-FreedBdd". */  

/*-----*/  

void FreePhysicalBdd (BddPtr)
    BDD_PTR BddPtr;
{

if (GLOB_BDD_WS->FreedBdd == BDD_PTR_NULL)
{
    GLOB_BDD_WS->FreedBdd = BddPtr;
    BddPtr->Next = BDD_PTR_NULL;
    return;
}

BddPtr->Next = GLOB_BDD_WS->FreedBdd;
GLOB_BDD_WS->FreedBdd = BddPtr;
```

```

}

/*
 *-----*/
/* FreeApplyCell */
/*-----*/
/* We free the current apply cell "ApplyCell" and store it in the list */
/* GLOB_BDD_WS-FreedApplyCell of the BDD work space. */
/*-----*/
void FreeApplyCell (ApplyCell)
    APPLY_CELL      ApplyCell;
{
    if (GLOB_BDD_WS->FreedApplyCell == NULL)
    {
        GLOB_BDD_WS->FreedApplyCell = ApplyCell;
        ApplyCell->Next = NULL;
        return;
    }

    ApplyCell->Next = GLOB_BDD_WS->FreedApplyCell;
    GLOB_BDD_WS->FreedApplyCell = ApplyCell;
}

/*
 *-----*/
/* SubBucketGarbageCollector */
/*-----*/
/* We collect the Dead bdd nodes of the Bucket "Cache[Index]". All the */
/* collected Bdd nodes are removed from the Bdd Cache and added in the */
/* list "GLOB_BDD_WS->FreedBdd" thru the function "FreePhysicalBdd". */
/*-----*/
void SubBucketGarbageCollector (Cache, Index)
    BLIST      Cache;
    int         Index;
{
    BDD_PTR    PreviousSSBucket;
    BDD_PTR    AuxSSBucket;
    BDD_PTR    SSBucket;
    int         i;
    BDD_STATUS   Status;

    for (i=0; i<BDD_PTR_SUB_CACHE_SIZE; i++)
    {
        PreviousSSBucket = BDD_PTR_NULL;
        SSBucket = (GetCache (Cache, Index)) ->Buckets[i];
        while (SSBucket != BDD_PTR_NULL)
        {
            if (GetBddPtrRefCount (SSBucket) == 0)
            {
                AuxSSBucket = SSBucket;
                if (PreviousSSBucket == BDD_PTR_NULL)
                {
                    GetCache (Cache, Index) ->Buckets[i] =
                        GetBddPtrNext (SSBucket);
                }
            }
        }
    }
}

```

```

        SSBucket = GetCache (Cache, Index) ->Buckets[i];
    }
else
{
    /* We skip the SSBucket by joining its previous */
/* SSBucket and its next one.                      */
    SetBddPtrNext (PreviousSSBucket,
                    GetBddPtrNext (SSBucket));
    SSBucket = GetBddPtrNext (PreviousSSBucket);
}

(void)FreePhysicalBdd (AuxSSBucket);

GetCache (Cache, Index) ->NbDeadBddNode--;
GLOB_BDD_WS->TotalBddDead--;
GetCache (Cache, Index) ->NbBddNode--;

/* no more Dead node in this Bucket so we can stop */
if (GetCache (Cache, Index) ->NbDeadBddNode == 0)
    break;
}
else
{
    PreviousSSBucket = SSBucket;
    SSBucket = GetBddPtrNext (SSBucket);
}
}

/* no more Dead node in this Bucket so we can stop */
if (GetCache (Cache, Index) ->NbDeadBddNode == 0)
    break;
}

}

/*-----
/* MaxDeadBddReached
*/
/*-----
/* If the number of dead nodes "dead" is more than 10% of the created */
/* bdd nodes "Created" then we collect the corresponding subckt. */
*/
int MaxDeadBddReached (Dead, Created)
{
    int      Dead;
    int      Created;
}

    if (10*Dead > Created)
        return (1);

    return (0);
}

/*-----
/* BddGarbageCollector
*/
/*-----
/* The garbage collector scans each bucket associated to each variable */

```

bbddmem.c

```
/* and if the "dead condition" is reached on this bucket then we collect*/
/* all the dead nodes in it. */
/*-----*/
void BddGarbageCollector ()
{
    int             i;
    BLIST          BddCache;

    BddCache = GLOB_BDD_WS->BddCache;

    for (i=0; i<BListSize (BddCache); i++)
    {
        if (MaxDeadBddReached (GetCache (BddCache, i)->NbDeadBddNode,
                               GetCache (BddCache, i)->NbBddNode))
            (void) SubBucketGarbageCollector (BddCache, i);
    }
}

/*-----*/
/* FreeBddRec */
/*-----*/
/* Recursive code of the main function "FreeBdd". We decrement the Ref */
/* count of the Bdd and if this one becomes 0 then we propagate recursi-*/
/* vely the Free on its sons since they have one less reference. We also*/
/* update the fact that one more Bdd is Dead for a special bucket of the*/
/* bdd cache. */
/*-----*/
BDD_STATUS FreeBddRec (Bdd)
    BDD Bdd;
{
    BDD_PTR BddPtr;
    BDD_STATUS Status;
    BLIST Cache;

    if (ConstantBdd (Bdd))
        return (BDD_OK);

    BddPtr = GetBddPtrFromBdd (Bdd);

    if (FrozenBddNode (BddPtr))
        return (BDD_OK);

    /* If the Reference count is 0 then we just decrement it. */
    if (GetBddPtrRefCount (BddPtr) > 1)
    {
        BddPtr->Header -= 1;

        return (BDD_OK);
    }

    if (GetBddPtrRefCount (BddPtr) == 0)           /* Should never occur */
    {
        fprintf (stderr, "BDD-WARNING: attempt to free an already freed
Bdd\n");
    }
}
```

bbddmem.c

```
    return (BDD_OK);
}

/* We pass from a Ref count = 1 to 0. So the Bdd becomes dead. We      */
/* may need to delete its corresponding BDD_CELL in the Bdd cache.   */
/* This will be performed during the garbage collecting phase. */
GLOB_BDD_WS->TotalBddDead++;
Cache = GLOB_BDD_WS->BddCache;
(GetCache (Cache,GetBddPtrIndex (BddPtr))->NbDeadBddNode)++;

/* We free the sons since they are not pointed anymore by the current */
/* Bdd that we are removing.                                         */
if ((Status = FreeBddRec (GetBddPtrEdge1 (BddPtr))))           */
    return (Status);

if ((Status = FreeBddRec (GetBddPtrEdge0 (BddPtr))))           */
    return (Status);

BddPtr->Header -= 1;

return (BDD_OK);
}

/*-----*/
/* FreeBdd                                */
/*-----*/
/* We free the BDD (either virtually or physically) and check if any */
/* garbage collection must be performed and invokes it if yes. Garbage */
/* is done if more than 10% of the BDD nodes are dead ones.          */
/*-----*/
BDD_STATUS FreeBdd (Bdd)
    BDD Bdd;
{
    BDD_STATUS     Status;

    Status = FreeBddRec (Bdd);

    /* If more than 50% of Bdd created are dead then we call the garbage */
    /* collector.                                                       */
    if (GLOB_BDD_WS->TotalBddDead*2 > GLOB_BDD_WS->TotalBddCreated)
    {
        (void) BddGarbageCollector ();
    }

    return (Status);
}

/*-----*/
/* CreateApplyCell                           */
/*-----*/
/* Creation of a basic apply cell in order to store apply informations. */
/* This object constitutes the basic object of any Apply caches. */
/*-----*/
BDD_STATUS CreateApplyCell (NewCell)
    APPLY_CELL *NewCell;
```

```

{
    OBJECT_PAGE    CurrentApplyCellPage;
    OBJECT_PAGE    NewPage;
    BDD_STATUS     Status;

    /* First we try to pick up a previous freed APPLY_CELL           */
    if ((*NewCell = GetFreedApplyCell()) != NULL)
        return (BDD_OK);

    CurrentApplyCellPage = GLOB_BDD_WS->CurrentApplyCellPage;

    /* If the current page is full then we create a new one.          */
    if (CurrentApplyCellPage->PageFull)
    {
        if ((Status = PageAllocate (APPLY_PAGE_SIZE, sizeof(APPLY_CELL_REC),
                                    &NewPage)))
            return (Status);

        NewPage->NextPage = CurrentApplyCellPage;
        GLOB_BDD_WS->CurrentApplyCellPage = NewPage;
        CurrentApplyCellPage = GLOB_BDD_WS->CurrentApplyCellPage;
    }

    *NewCell = (APPLY_CELL) CurrentApplyCellPage->NextFreeObject;
    if (CurrentApplyCellPage->NextFreeObject == CurrentApplyCellPage-
>LastObject)
        CurrentApplyCellPage->PageFull = 1;
    else
        CurrentApplyCellPage->NextFreeObject += sizeof(APPLY_CELL_REC);

    GLOB_BDD_WS->TotalApplyCreated++;

    return (BDD_OK);
}

/*
 * CreateApplyCache
 */
/*
 * Creation of a cache (hash table) of Apply cells.
 */
BDD_STATUS CreateApplyCache (NewCache)
    APPLY_CACHE    *NewCache;
{

    if ((*NewCache = (APPLY_CACHE) BDD_CALLOC (1,
                                                sizeof(APPLY_CACHE_REC))) == NULL)
        return (BDD_MEMORY_FULL);

    if (((*NewCache)->Buckets = (APPLY_CELL*) BDD_CALLOC (APPLY_CACHE_SIZE,
                                                          sizeof (APPLY_CELL))) == NULL)
        return (BDD_MEMORY_FULL);

    (*NewCache)->CacheSize = APPLY_CACHE_SIZE;
}

```

```

        return (BDD_OK);
    }

/*-----*/
/* CreateBddCache */
/*-----*/
/* Creation of the unique Bdd table/cache */
/*-----*/
BDD_STATUS CreateBddCache (MaxVar, NewCache)
    int          MaxVar;
    BLIST      *NewCache;
{
    int          i;

    if (BListCreateWithSize (MaxVar, NewCache))
        return (BDD_MEMORY_FULL);

    return (BDD_OK);
}

/*-----*/
/* CreateBddVar */
/*-----*/
/* Creation of a primary Var requiring its name as parameter. A unique */
/* Id is given to this var. An associated Bdd is contructed and added in */
/* the unique Bdd table. */
/* CAUTION: the name is pointed by one of the field of the 'Var' so take */
/* care when you free it to update the name field of 'Var'. */
/*-----*/
BDD_STATUS CreateBddVar (Name, Var)
    char          *Name;
    BDD_VAR     *Var;
{
    BDD_PTR  NewBdd;
    BDD_STATUS   Status;
    BDD_CACHE   NewCache;

    ++(GLOB_BDD_WS->UniqueId);

    /* The current new index is greater or equal than the max allowed */
    if ((UInt32) (GLOB_BDD_WS->UniqueId) >= BDD_MAXINDEX)
        return (BDD_MAX_VAR_ID_REACHED);

    if ((*Var = (BDD_VAR) BDD_CALLOC (1, sizeof(BDD_VAR_REC))) == NULL)
        return (BDD_MEMORY_FULL);

    /* Adding the new variable in the main list of variables. */
    /* Its index is by construction the current value of UniqueId. */
    if (BListAddElt (GLOB_BDD_WS->Var, (int) (*Var)))
        return (BDD_MEMORY_FULL);

    /* we create a new cache for this variable in the list BddCache */
    if ((NewCache = (BDD_CACHE) malloc (1, sizeof(BDD_CACHE_REC))) == NULL)
        return (BDD_MEMORY_FULL);
    if (BListAddElt (GLOB_BDD_WS->BddCache, (int) NewCache))
        return (BDD_MEMORY_FULL);
}

```

bbddmem.c

```
if ((NewCache->Buckets = (BDD_PTR*)BDD_CALLOC (BDD_PTR_SUB_CACHE_SIZE,
                                                sizeof(BDD_PTR)) == NULL)
    return (BDD_MEMORY_FULL);
NewCache->CacheSize = BDD_PTR_SUB_CACHE_SIZE;

/* Creating and adding in the Bdd cache. */
if ((Status = AddOrGetBddInCache (GLOB_BDD_WS->BddCache,
                                  GLOB_BDD_WS->UniqueId,
                                  BDD_1, BDD_0, &NewBdd)))
    return (Status);

(void)bdd_freeze (NewBdd); /* This Bdd is frozen */
(*Var)->Name = Name;
(*Var)->Bdd = GetBddFromBddPtr (NewBdd);
(*Var)->Id = GLOB_BDD_WS->UniqueId;

return (BDD_OK);

}

/*-----
/* SystemMemoryGet
*/
/*-----*/
/* get info on memory usage, works only for SUN machines */
/*-----*/
int SystemMemoryGet ()
{
#ifndef MEMORY
    struct mallinfo Info;

    Info = mallinfo ();
    return (Info.uordblks + Info.usmblks);
#else
    return (0);
#endif

}

/*-----
/* FreeBddWorkSpace
*/
/*-----*/
/* This function frees completely the BDD work space and guarantees a */
/* 0-memory leak between the call of "InitBddWorkSpace" and itself. */
/*-----*/
void FreeBddWorkSpace (Ws)
    BDD_WS    Ws;
{
    int          i;
    OBJECT_PAGE  CurrentPage;
    OBJECT_PAGE  PreviousPage;
    APPLY_CELL   ApplyCell;
    APPLY_CELL   PreviousApplyCell;
    BLIST       Cache;

    free ((char*)Ws->WorkSpaceName);
    for (i=1; i<=Ws->UniqueId; i++)
```

```

{
    if (Ws->Var)
    {
        free ((char*)GetWsBddVar(Ws,i));
    }
}
BListQuickDelete (&(Ws->Var));

CurrentPage = Ws->CurrentApplyCellPage;
while (CurrentPage)
{
    PreviousPage = CurrentPage;
    CurrentPage = CurrentPage->NextPage;

    free ((char*)PreviousPage->FirstObject);
    free ((char*)PreviousPage);
}

CurrentPage = Ws->CurrentBddPage;
while (CurrentPage)
{
    PreviousPage = CurrentPage;
    CurrentPage = CurrentPage->NextPage;

    free ((char*)PreviousPage->FirstObject);
    free ((char*)PreviousPage);
}

free ((char*)Ws->OrCache->Buckets);
free ((char*)Ws->OrCache);
free ((char*)Ws->AndCache->Buckets);
free ((char*)Ws->AndCache);
free ((char*)Ws->ImpCache->Buckets);
free ((char*)Ws->ImpCache);
free ((char*)Ws->EqCache->Buckets);
free ((char*)Ws->EqCache);
free ((char*)Ws->RstCache->Buckets);
free ((char*)Ws->RstCache);
free ((char*)Ws->JoinCache->Buckets);
free ((char*)Ws->JoinCache);

Cache = Ws->BddCache;
for (i=1; i<BListSize (Cache); i++)
{
    free ((char*) (GetCache (Cache,i)->Buckets));
}
BListQuickDelete (&Cache);

free ((char*)Ws);
}

/*-----*/

```

blist.c

```
/*-----*/
/*          */
/*      File:      blist.c           */
/*      Version:    1.1             */
/*      Modifications: -          */
/*      Documentation: -         */
/*          */
/*      Class: none            */
/*      Inheritance:          */
/*          */
/*          */
/*-----*/
#include <stdio.h>
#include <string.h>
#include <malloc.h>

#include "blist.h"

/* -----
/* BListCreateWithSize
/* -----
/* This function creates a list of "size" elements. It returns 0 if
/* everything is OK and 1 if there was something wrong. Blist will be
/* the new allocated list.
/* -----
int BListCreateWithSize (Size, Blist)
    int          Size;
    BLIST      *Blist;
{
    BLIST *r;

    if ((*Blist = (BLIST)calloc (1, sizeof (BLIST_REC))) == NULL)
        return (1);

    (*Blist)->Size = (Size > 0 ? Size : 1);
    (*Blist)->NbElt = 0;

    if (((*Blist)->Adress =
                (int*)calloc ((*Blist)->Size, sizeof(int))) == NULL)
    {
        free ((char*)(*Blist));
        *Blist = NULL;
        return (1);
    }

    return (0);
}

/* -----
/* BListCreate
/* -----
```

blist.c

```
/* Creates a default BList with a default size of 2 */  
/* ----- */  
int BListCreate (Blist)  
    BLIST *Blist;  
{  
    return (BListCreateWithSize (2, Blist));  
}  
  
/* ----- */  
/* BListQuickDelete */  
/* ----- */  
void BListQuickDelete (L)  
    BLIST *L;  
{  
    if (*L)  
    {  
        free ((char *) (*L) ->Adress);  
        free ((char *) *L);  
        *L = NULL;  
    }  
}  
  
/* ----- */  
/* BListDelShift */  
/* ----- */  
/* Returns 1 if failed and 0 if success. */  
/* ----- */  
int BListDelShift (L, Rank)  
    BLIST L;  
    int Rank;  
{  
    int i;  
  
    if (Rank <= 0)  
        return (0);  
  
    if (Rank > L->NbElt)  
    {  
        return (1);  
    }  
  
    for (i = Rank; i < L->NbElt; i++)  
        (L->Adress) [i-1] = (L->Address) [i];  
    L->NbElt--;  
  
    return (0);  
}  
  
/* ----- */  
/* BListDelInsert */  
/* ----- */  
/* Returns 1 if failed and 0 if success. */  
/* ----- */  
int BListDelInsert (L, Rank)  
    BLIST L;  
    int Rank;
```

blist.c

```
{  
    if (Rank <= 0)  
        return (0);  
  
    if (Rank > L->NbElt)  
    {  
        return (1);  
    }  
  
    (L->Adress) [Rank - 1] = (L->Adress) [L->NbElt - 1];  
    L->NbElt--;  
  
    return (0);  
}  
  
/* ----- */  
/* BListAddElt */  
/* ----- */  
int BListAddElt (L, Elt)  
    BLIST      L;  
    int       Elt;  
{  
  
    if (L->NbElt >= L->Size)  
    {  
        L->Size *= 2;  
        if ((L->Adress = (int*) realloc ((char*) (L->Adress),  
                                         (unsigned) (L->Size) * sizeof (int))) == NULL)  
            return (1);  
    }  
  
    (L->Adress) [L->NbElt] = Elt;  
    L->NbElt++;  
  
    return (0);  
}  
  
/* ----- */  
/* IntIdentical */  
/* ----- */  
int IntIdentical (I1, I2)  
    int I1, I2;  
{  
    return (I1 == I2);  
}  
  
/* ----- */  
/* StringIdentical */  
/* ----- */  
int StringIdentical (S1, S2)  
    char     *S1;  
    char     *S2;  
{  
    return (!strcmp (S1, S2));  
}
```

blist.c

```
/* ----- */
/* BListMemberOfList */
/* ----- */
int BListMemberOfList (L1, Elt, ElemIdentical)
    BLIST      L1;
    int        Elt;
    IntFctPtr ElemIdentical;
{
    int        i;
    int        *stop1;
    int    *elem1;

    if (L1 == NULL)
        return (0) ;

    i = 1 ;
    for (stop1 = (elem1 = L1->Adress) + L1->NbElt;
                     elem1 < stop1; elem1++, i++)
        if ((*ELEMIDENTICAL) (*elem1, Elt))
            return (i) ;

    return (0);
}

/* ----- */
/* BListElemFree */
/* ----- */
void BListElemFree (Elem)
    int    *Elem;
{
    free ((char *) *Elem);
    *Elem = (int)NULL;
}

/* ----- */
/* BListDelete */
/* ----- */
void BListDelete (L, ElemDelete)
    BLIST      *L;
    VoidFctPtr   ElemDelete;
{
    int        *stop1;
    int    *elem1;

    if (*L)
    {
        for (stop1 = (elem1 = (*L)->Adress) + (*L)->NbElt; elem1 < stop1;)
            ElemDelete (elem1++);
        BListQuickDelete (L);
    }
}

/* ----- */
/* BListInsertInList */
/* ----- */
```

blist.c

```
/* ----- */
int BListInsertInList (L, Elt, Rank)
    BLIST      L;
    int        Elt;
    int        Rank;
{
    int i;

    if ((Rank > BListSize(L) + 1) || (Rank <= 0))
    {
        return (1);
    }
    else
    {
        BListAddElt (L, Elt);
        for (i = L->NbElt - 1; i >= Rank; i--)
            (L->Adress)[i] = (L->Adress)[i - 1];
        (L->Adress)[Rank-1] = Elt;

        return (0);
    }
}

/* ----- */
/* BListCopyNoEltCr                                */
/* ----- */
/* Copy a list of elements but do not copy physically the elements.      */
/* Returns 1 if failure and 0 if success.                                     */
/* ----- */
int BListCopyNoEltCr (L, Lcopy)
    BLIST      L;
    BLIST      *Lcopy;
{
    int        i;

    if (L == NULL)
    {
        *Lcopy = NULL;
        return (0);
    }

    if (BListCreate (Lcopy))
        return (1);

    for (i = 0; i < L->NbElt; i++)
        if (BListAddElt (*Lcopy, (int) L->Adress[i]))
            return (1);

    return (0);
}

/* ----- */
/* BListUnionNoEltCr                                */
/* ----- */
int BListUnionNoEltCr (L1, L2, UnionList)
```

blist.c

```
BLIST      L1;
BLIST      L2;
BLIST      *UnionList;
{
    int      i;

    if (L1 == NULL)
    {
        if (BListCopyNoEltCr (L2, UnionList))
            return (1);
    }
    else
    {
        if (BListCopyNoEltCr (L1, UnionList))
            return (1);

        for (i = 0; i < BListSize (L2); i++)
        {
            if (!BListMemberOfList (L1, BListElt (L2, i), IntIdentical))
                if (BListAddElt (*UnionList, BListElt (L2, i)))
                    return (1);
        }
    }
    return (0);
}

/* -----
/* BListIntersectionNoEltCr                      */
/* ----- */
/* returns the intersection of L1 and L2 according to the comparison   */
/* function between two elements if no intersection returns NULL  */
/* ----- */
int BListIntersectionNoEltCr (L1, L2, ElemIdentical, InterList)
    BLIST      L1;
    BLIST      L2;
    IntFctPtr    ElemIdentical;
    BLIST      *InterList;
{
    int      i;

    if ((L1 == NULL) || (L2 == NULL))
    {
        *InterList = NULL;
        return (0);
    }

    if (BListCreate (InterList))
        return (1);

    for (i = 0; i < BListSize (L1); i++)
        if (BListMemberOfList (L2, BListElt (L1, i), ElemIdentical))
            if (BListAddElt (*InterList, BListElt (L1, i)))
                return (1);
```

blist.c

```
if (BListSize (*InterList) == 0)
{
    BListQuickDelete (InterList);
    *InterList = NULL;
    return (0);
}

return (0);
}

/* -----
/* BListAppend                               */
/* ----- */
int BListAppend (L1, L2)
    BLIST    L1;
    BLIST    *L2;
{
    int      i;

    if (!(*L2))
        return (0);

    for (i=0; i<BListSize (*L2); i++)
    {
        if (BListAddElt (L1, BListElt (*L2, i)))
            return (1);
    }
    BListQuickDelete (L2);

    return (0);
}

/* -----
/* BListAddNoEltCr                          */
/* ----- */
/* Adds L2 to L1 without destroying L2       */
/* ----- */
int BListAddNoEltCr (L1, L2)
    BLIST    L1;
    BLIST    L2;
{
    int      i;

    if (L2)
    {
        for (i=0; i<BListSize (L2); i++)
        {
            if (BListAddElt (L1, BListElt (L2, i)))
                return (1);
        }
    }

    return (0);
}
```

blist.c

```
/* ----- */
/* BListContextualIdentical */ /* ----- */
/* ----- */
int BListContextualIdentical (L1, L2, ElemIdentical, Context)
    BLIST    L1;
    BLIST    L2;
    IntFctPtr    ElemIdentical;
    int        *Context;
{
    int        *elem1;
    int        *elem2;
    int        *stop1;
    int        *stop2;

    stop1 = (elem1 = L1->Adress) + L1->NbElt;
    stop2 = (elem2 = L2->Adress) + L2->NbElt;
    do
    {
        while (1)
        {
            if (elem2 == stop2)
                return (0);

            if (ELEMIDENTICAL (*elem2++, *elem1, Context))
                break;
        }
        elem2 = L2->Adress;
    }
    while (++elem1 < stop1);

    elem1 = L1->Adress;
    elem2 = L2->Adress;
    do
    {
        while (1)
        {
            if (elem1 == stop1)
                return (0);

            if (ELEMIDENTICAL (*elem1++, *elem2, Context))
                break;
        }
        elem1 = L1->Adress;
    }
    while (++elem2 < stop2);

    return (1);
}

/* ----- end of file ----- */

```

blist.e

```
/*----- */
/*          blist.e
/* Version:      1.1
/* Modifications: -
/* Documentation: -
/*          none
/* Inheritance:
/*----- */

/* -----
/* BListCreateWithSize
/* -----
/* This function creates a list of "size" elements. It returns 0 if
/* everything is OK and 1 if there was something wrong. Blist will be
/* the new allocated list.
/* -----
extern int BListCreateWithSize /* Size, Blist */;

/*
    int           Size;
    BLIST        *Blist;
*/

/* -----
/* BListCreate
/* -----
/* Creates a default BList with a default size of 10
/* -----
extern int BListCreate /* Blist */;

/*
    BLIST        *Blist;
*/

/* -----
/* BListQuickDelete
/* -----
extern void BListQuickDelete /* L */;

/*
    BLIST *L;
*/

/* -----
/* BListDelShift
/* -----
/* Returns 1 if failed and 0 if success.
/* -----
extern int BListDelShift /* L, Rank */;

/*
    BLIST      L;
    int         Rank;
*/
```

blist.e

```
/* ----- */
/* BListDelInsert */ */
/* ----- */
/* Returns 1 if failed and 0 if success. */
/* ----- */
extern int BListDelInsert /* L, Rank */;

/*
    BLIST      L;
    int        Rank;
*/

/* ----- */
/* BListAddElt */ */
/* ----- */
extern int BListAddElt /* L, Elt */;

/*
    BLIST      L;
    int        Elt;
*/

/* ----- */
/* IntIdentical */ */
/* ----- */
extern int IntIdentical /* I1, I2 */;

/*
    int I1, I2;
*/

/* ----- */
/* StringIdentical */ */
/* ----- */
extern int StringIdentical /* S1, S2 */;

/*
    char      *S1;
    char      *S2;
*/

/* ----- */
/* BListMemberOfList */ */
/* ----- */
extern int BListMemberOfList /* L1, Elt, ElemIdentical */;

/*
    BLIST      L1;
    int        Elt;
    IntFctPtr ElemIdentical;
*/

/* ----- */
/* BListElemFree */ */
/* ----- */
extern void BListElemFree /* Elem */;

/*
    int      *Elem;
*/

/* ----- */
```

blist.e

```
/* BListDelete */  
/* ----- */  
extern void BListDelete /* L, ElemDelete */;  
/*  
    BLIST      *L;  
    VoidFctPtr   ElemDelete;  
*/  
  
/* ----- */  
/* BListInsertInList */  
/* ----- */  
extern int BListInsertInList /* L, Elt, Rank */;  
/*  
    BLIST      L;  
    int       Elt;  
    int       Rank;  
*/  
  
/* ----- */  
/* BListCopyNoEltCr */  
/* ----- */  
/* Copy a list of elements but do not copy physically the elements. */  
/* Returns 1 if failure and 0 if success. */  
/* ----- */  
extern int BListCopyNoEltCr /* L, Lcopy */;  
/*  
    BLIST      L;  
    BLIST      *Lcopy;  
*/  
  
/* ----- */  
/* BListUnionNoEltCr */  
/* ----- */  
/* returns the union of L1 and L2 according to the comparison function */  
/* between two elements. Add elemnts of L2 to L1 which is returned */  
/* ----- */  
extern int BListUnionNoEltCr /* L1, L2, ElemIdentical, UnionList */;  
/*  
    BLIST      L1;  
    BLIST      L2;  
    IntFctPtr   ElemIdentical;  
    BLIST      *UnionList;  
*/  
  
/* ----- */  
/* BListIntersectionNoEltCr */  
/* ----- */  
/* returns the intersection of L1 and L2 according to the comparison */  
/* function between two elements if no intersection returns NULL */  
/* ----- */  
extern int BListIntersectionNoEltCr /* L1, L2, ElemIdentical, InterList */;  
/*  
    BLIST      L1;  
    BLIST      L2;  
    IntFctPtr   ElemIdentical;  
    BLIST      *InterList;
```

blist.e

```
*/  
/* ----- */
```

blist.h

```
/*-----*/
/*
 *      File:          blist.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#ifndef BLIST_H
#define BLIST_H

typedef struct BLIST_STRUCT {
    int      Size;
    int      NbElt ;
    int      *Adress;
}
BLIST_REC, *BLIST ;

typedef int (*IntFctPtr) () ;
typedef int *(*PtIntFctPtr) () ;
typedef int (*EltFctPtr) () ;
typedef void (*VoidFctPtr) () ;

/*
 *-----*/
/* DEFINE */
/*-----*/
#define     BSize(L)           ((L)->NbElt)
#define     BListAddress(L)    ((L)->Adress)
#define     BListSize(L)       (((L)==NULL)?0:(L)->NbElt)
#define     BListElt(L,Index)   (((L)->Adress)[(Index)])

/*-----*/
EOF
/*-----*/
#endif
```

gnl.h

```
/*-----*/
/*
 *      File:          gnl.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*-----*/
#ifndef GNL_H
#define GNL_H

/*-----*/
/* DEFINE
/*-----*/
#define GNL_TOOL_NAME      "HUBBLE-RTL"
#define GNL_MAPIT_VERSION   "12.0 Alpha"

#define HASH_TABLE_NAMES_SIZE      500001
#define SMALL_HASH_TABLE_NAMES_SIZE 100
#define HASH_TABLE_COMPO_NAMES_SIZE 1000
#define HASH_LIST_PATH_COMPO_SIZE  1

/*-----*/
/* GNL_STATUS
/*-----*/
typedef enum {
    GNL_OK,
    GNL_VAR_EXISTS,
    GNL_VAR_NOT_EXISTS,
    GNL_BAD_ARG_NUMBER,
    GNL_UNKNOWN_GATE,
    GNL_VAR_REDEFINED,
    GNL_BAD_CLOCK_DEFINITION,
    GNL_BAD_GNL,
    GNL_MULTISOURCE,
    GNL_MAP_ERROR,
    GNL_CYCLE_DETECTED,
    GNL_BAD_INSTANCE_INTERFACE,
    GNL_CANNOT_OPEN_INPUTFILE,
    GNL_CANNOT_OPEN_OUTFILE,
    GNL_CANNOT_OPEN_LIBRARY,
    GNL_CANNOT_FIND_TOP_LEVEL,
    GNL_ERROR_LIBRARY_READING,
    GNL_BAD_FSM_FOR_SYNTHESIS,
    GNL_FSM_PARSE_ERROR,
    GNL_BAD_INPUT_FORMAT,
    GNL_MEMORY_FULL,
    GNL_UNRESOLVED,
    GNL_UNRESOLVED_AT_XOR_BUILD,
    GNL_UNRESOLVED_AT_BUILD_CUTPOINT,
    GNL_UNRESOLVED_AT_EXTRACTION,
    GNL_EQUIVALENT,
    GNL_EQUIVALENT_AT_XOR_BUILD,
    GNL_EQUIVALENT_PHASE0,
```

```

GNL_EQUIVALENT_PHASE1,
GNL_EQUIVALENT_PHASE2,
GNL_ERROR_DETECTED,
GNL_ERROR_DETECTED_PHASE0,
GNL_ERROR_DETECTED_PHASE1,
GNL_ERROR_DETECTED_PHASE2,
GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS,
GNL_ERROR_DETECTED_PHASE1_ONLY_INPUTS,
GNL_ERROR_DETECTED_PHASE2_ONLY_INPUTS,
GNL_ERROR_DETECTED_AT_XOR_BUILD
} GNL_STATUS;

/*-----*/
/* GNL_TYPE */
/*-----*/
typedef enum {
    GNL_USER,
    GNL_CELL
} GNL_TYPE;

/*-----*/
/* GNL_VAR_DIR */
/*-----*/
/* CAUTION !
 * The char value of these macros must be strictly greater than the
 * values of the macro of the type GNL_OP.
 * Please keep the order and keep GNL_VAR_LOCAL as lowest char value
 * CAUTION: do not touch the macro below and char values associated to
 * GNL_VAR_DIR and GNL_OP !
 *-----*/
typedef char GNL_VAR_DIR;
#define GNL_VAR_LOCAL      'a'   /* ascii 97           */
#define GNL_VAR_LOCAL_WIRING 'b'   /* ascii 98           */
#define GNL_VAR_INPUT       'c'   /* ascii 99           */
#define GNL_VAR_INOUT        'd'   /* ascii 100          */
#define GNL_VAR_OUTPUT       'e'   /* ascii 101          */
#define GNL_VAR_LOCAL_UNDRIVEN 'f'   /* ascii 102          */
#define GNL_VAR_LOCAL_DONTCARE 'g'   /* ascii 103          */
#define GNL_VAR_FSM_FUNCTION 'h'   /* ascii 104          */
#define GNL_STATE_VAR        'i'   /* ascii 105          */
#define GNL_OUT_STATE_VAR     'j'   /* ascii 106          */
#define GNL_VAR_FSM_BLACK_BOX 'k'   /* ascii 107          */
#define GNL_VAR_FSM_BLACK_BOX_INOUT 'l'   /* ascii 108          */

/*-----*/
/* GNL_VAR_TYPE */
/*-----*/
/* Var can be of two types: GNL_VAR_ORIGINAL if it has been defined
 * originally by the user or GNL_VAR_EXPANSED if it has been generated
 * internally by a Bus expansion process.
 *-----*/
typedef char GNL_VAR_TYPE;
#define GNL_VAR_ORIGINAL     'O'   /* Originally from User */
#define GNL_VAR_EXPANSED      'E'   /* From a bus expansion */
#define GNL_VAR_INTERNAL       'I'   /* created internally */

```

```

/*
 *-----*/
/* GNL_FORM_OUTPUT */ 
/*-----*/
/* Forms of outputs for any sequential component. */
/*-----*/
typedef char GNL_FORM_OUTPUT;
#define GNL_Q_QBAR      'p'   /* It has two outputs Q and !Q */
#define GNL_Q           'q'   /* It has one output Q */
#define GNL_QBAR        'r'   /* It has one output !Q */

/*
 *-----*/
/* GNL_TRIG */ 
/*-----*/
/* Signals behaviors in GNL */
typedef enum {
    GNL_NO_TRIG,
    GNL_RISING,
    GNL_FALLING,
    GNL_LOW_LEVEL,
    GNL_HIGH_LEVEL
} GNL_TRIG;

/*
 *-----*/
/* GNL_OP */ 
/*-----*/
/* CAUTION ! */
/* The char value of these macros must be strictly less than the values */
/* of the macro of the type GNL_VAR_DIR */
/* CAUTION: do not touch the macro below and char values associated to */
/* GNL_VAR_DIR and GNL_OP ! */
/*-----*/
typedef char GNL_OP;
#define GNL_AND         'A'   /* ascii 65 */
#define GNL_NAND        'B'   /* ascii 66 */
#define GNL_OR          'C'   /* ascii 67 */
#define GNL_NOR         'D'   /* ascii 68 */
#define GNL_NOT         'E'   /* ascii 69 */
#define GNL_XOR         'F'   /* ascii 70 */
#define GNL_XNOR        'G'   /* ascii 71 */
#define GNL_WIRE         'H'   /* ascii 72 */
#define GNL_VARIABLE     'I'   /* ascii 73 */
#define GNL_CONSTANTE    'J'   /* ascii 74 */
/* Special operator in netlist. Only used to concatante signals for an */
/* actual port in a component instantiation (cf. GNL_ASSOC) */
#define GNL_CONCAT       'K'   /* ascii 75 */

/*
 *-----*/
/* GNL_SEQUENTIAL_OP */ 
/*-----*/
/* Operators for sequential components. */
/*-----*/
typedef enum {
    GNL_DFF,      /* Flip-Flop with no reset nor set. */
}

```

```

GNL_DFFX,      /* Flip-Flop with set/reset, X value when both */
GNL_DFF0,      /* Flip-Flop with set/reset, 0 value when both */
GNL_DFF1,      /* Flip-Flop with set/reset, 1 value when both */
GNL_LATCH,     /* Latch with no reset nor set. */
GNL_LATCHX,    /* Latch with set/reset, X value when both */
GNL_LATCH0,    /* Latch with set/reset, 0 value when both */
GNL_LATCH1,    /* Latch with set/reset, 1 value when both */
} GNL_SEQUENTIAL_OP;

/*-----*/
/* GNL_COMPONENT_TYPE */
/*-----*/
typedef char GNL_COMPONENT_TYPE;
#define GNL_SEQUENTIAL_COMPO 'S'
#define GNL_TRISTATE_COMPO 'T'
#define GNL_MACRO_COMPO 'M'
#define GNL_USER_COMPO 'U'
#define GNL_BUF_COMPO 'V'
#define GNL_BIDIR_COMPO 'W'

/*-----*/
/* GNL_VAR_STRUCT */
/*-----*/
/* NOTE: a GNL_VAR with Msb = Lsb = -1 means a Var with a undefined
range. */
/*-----*/
typedef struct GNL_VAR_STRUCT
{
    /* the first field must be 'GNL_VAR_DIR' Dir; */
    GNL_VAR_DIR Dir;      /* GNL_VAR_LOCAL, GNL_VAR_INPUT, ... */
    /* */

    GNL_VAR_TYPE Type;    /* Expanded, original or internal */

    short int Msb;        /* Most significant bit */
    short int Lsb;        /* Least significant bit */
    char *Name;
    int Id;               /* Unique Id identifying the Var. */
    int Rank;             /* Rank in the list
                           /* {Inputs+Outputs+Locals} */
    int *Function; /* The function the var points on */
    BLIST Dads;          /* Elts of type <GNL_NODE>. List of
                           /* nodes which point on this Variable. */
    /* */

    char *Location; /* Location */

    int Tag; /* Tag for traversal algorithms */
    float OutCapa; /* for NL Delay mapping. */
    void *Hook; /* to hook any particular structure */
}
GNL_VAR_REC, *GNL_VAR;

/* Usefull to determine if a presupposed GNL_VAR object is a true */
/* GNL_VAR. (because GNL_NODE and GNL_VAR are sometimes unified like in */

```

gnl.h

```
/* the GNL_ASSOC structure for field Actual). */  
/* CAUTION: do not touch the macro below and char values associated to */  
/* GNL_VAR_DIR and GNL_OP ! */  
  
#define GnlVarIsVar(v) (((v)->Dir) >= GNL_VAR_LOCAL)  
  
#define GnlVarDir(v) (((v)->Dir))  
#define GnlVarType(v) (((v)->Type))  
#define GnlVarMsb(v) (((v)->Msb))  
#define GnlVarLsb(v) (((v)->Lsb))  
#define GnlVarName(v) (((v)->Name))  
#define GnlVarId(v) (((v)->Id))  
#define GnlVarRank(v) (((v)->Rank))  
#define GnlVarDriveFuncId(v) (((v)->Rank)) /* in fsm read mode */  
#define GnlVarDistance(v) (((v)->Rank)) /* in verification mode */  
#define GnlVarFunction(v) ((GNL_FUNCTION)((v)->Function))  
#define GnlVarDads(v) (((v)->Dads))  
#define GnlVarLocation(v) (((v)->Location))  
#define GnlVarTag(v) (((v)->Tag))  
#define GnlVarOutCapa(v) (((v)->OutCapa))  
#define GnlVarHook(v) (((v)->Hook))  
  
#define SetGnlVarDir(v,d) (((v)->Dir = d))  
#define SetGnlVarType(v,t) (((v)->Type = t))  
#define SetGnlVarMsb(v,m) (((v)->Msb = m))  
#define SetGnlVarLsb(v,l) (((v)->Lsb = l))  
#define SetGnlVarName(v,n) (((v)->Name = n))  
#define SetGnlVarId(v,i) (((v)->Id = i))  
#define SetGnlVarRank(v,r) (((v)->Rank = r))  
#define SetGnlVarDriveFuncId(v,r) (((v)->Rank = r)) /* fsm mode */  
#define SetGnlVarDistance(v,r) (((v)->Rank = r)) /* verif.mode */  
*/  
#define SetGnlVarFunction(v,f) (((v)->Function = (int*)f))  
#define SetGnlVarDads(v,d) (((v)->Dads = d))  
#define SetGnlVarLocation(v,ln) (((v)->Location = ln))  
#define SetGnlVarTag(v,t) (((v)->Tag = t))  
#define SetGnlVarOutCapa(v,t) (((v)->OutCapa = t))  
#define SetGnlVarHook(v,h) (((v)->Hook = (void*)h))  
  
/* ----- */  
/* GNL_CUBE_STRUCT */  
/* ----- */  
typedef struct GNL_CUBE_STRUCT  
{  
    unsigned *High;  
    unsigned *Low;  
}  
    GNL_CUBE_REC, *GNL_CUBE;  
  
#define GnlCubeHigh(c) (((c)->High))  
#define GnlCubeLow(c) (((c)->Low))  
  
#define SetGnlCubeHigh(c,h) (((c)->High = h))  
#define SetGnlCubeLow(c,l) (((c)->Low = l))  
  
/* ----- */  
/* GNL_NODE_STRUCT */  
/* ----- */
```

gnl.h

```
/* This structure represents the Node of a Boolean Tree. General Node      */
/* operators are classical Boolean operators like GNL_AND, GNL_OR or      */
/* GNL_NOT. They can be also GNL_VARIABLE, then the list of sons is the      */
/* pointer to the GNL_VAR variable.                                         */
/* It can be otherwise a GNL_CONSTANT representing either the              */
/* Boolean values: 0 (The field Sons is 0) or 1 (The field Sons is 1).    */
/* ----- */
#define SEGMENT_NODE_SIZE          100

typedef struct GNL_NODE_STRUCT
{
    /* The first field must be 'GNL_OP'                                Op; */
    GNL_OP           Op;

    BLIST            Sons;           /* List of <GNL_NODE>      */
    BLIST            Dads;          /* List of <GNL_NODE>      */
    int              LineNumber;
    int              Tag;
    void             *OriginalCompo;
    void             *Hook ;
}
GNL_NODE_REC, *GNL_NODE;

#define GnlNodeTag(n)          (((n)->Tag))
#define GnlNodeOp(n)           (((n)->Op))
#define GnlNodeSons(n)          (((n)->Sons))
#define GnlNodeDads(n)          (((n)->Dads))
#define GnlNodeLineNumber(n)    (((n)->LineNumber))
#define GnlMapNodeInfoOriginalCompo(n) ((((n)->OriginalCompo)))
#define GnlNodeHook(n)          (((n)->Hook))

#define SetGnlNodeTag(n,t)      (((n)->Tag = t))
#define SetGnlNodeOp(n,o)       (((n)->Op = o))
#define SetGnlNodeSons(n,s)     (((n)->Sons = s))
#define SetGnlNodeDads(n,d)    (((n)->Dads = d))
#define SetGnlNodeLineNumber(n,d) (((n)->LineNumber = d))
#define SetGnlMapNodeInfoOriginalCompo(n,d) ((((n)->OriginalCompo = d)))
#define SetGnlNodeHook(n,h)     (((n)->Hook = h))

/* ----- */
/* GNL_FUNCTION_STRUCT                                                 */
/* ----- */
typedef struct GNL_FUNCTION_STRUCT
{
    GNL_VAR   FctVar; /* Variable corresponding to the function      */
    GNL_NODE  OnSet;
    BLIST     ListOnCubes;
    GNL_NODE  DCSet;
    BLIST     ListDCCubes;
    GNL_NODE  ZSet;
}
GNL_FUNCTION_REC, *GNL_FUNCTION;

#define GnlFunctionVar(f)        (((f)->FctVar))
#define GnlFunctionOnSet(f)      (((f)->OnSet))
#define GnlFunctionOnSetCubes(f) (((f)->ListOnCubes))
#define GnlFunctionDCSet(f)      (((f)->DCSet))
```

gnl.h

```
#define GnlFunctionDCSetCubes(f)      (((f)->ListDCCubes))
#define GnlFunctionZSet(f)            (((f)->ZSet))

#define SetGnlFunctionVar(f,v)        (((f)->FctVar = v))
#define SetGnlFunctionOnSet(f,o)     (((f)->OnSet = o))
#define SetGnlFunctionOnSetCubes(f,c) (((f)->ListOnCubes = c))
#define SetGnlFunctionDCSet(f,dc)    (((f)->DCSet = dc))
#define SetGnlFunctionDCSetCubes(f,c) (((f)->ListDCCubes = c))
#define SetGnlFunctionZSet(f,dc)     (((f)->ZSet = dc))

/* -----
/* GNL_CONSTRAINT
/* -----
typedef struct GNL_CONSTRAINT_STRUCT
{
    GNL_NODE  Node;
    int        Value;
}
GNL_CONSTRAINT_REC, *GNL_CONSTRAINT;

#define GnlConstraintNode(c)          (((c)->Node))
#define GnlConstraintValue(c)         (((c)->Value))

#define SetGnlConstraintNode(c,n)    (((c)->Node = n))
#define SetGnlConstraintValue(c,v)   (((c)->Value = v))

/* -----
/* GNL_ASSOC
/* -----
typedef struct GNL_ASSOC_STRUCT
{
    void          *FormalPort,
    GNL_VAR ActualPort; /* It is generally a GNL_VAR object but */
                         /* it can be also a GNL_NODE with op. */
                         /* set to GNL_CONCAT */
                         /* This can be determined thru the first*/
                         /* field of the structure pointed by */
                         /* 'ActualPort'. */
    void          *Hook;
}
GNL_ASSOC_REC, *GNL_ASSOC ;

#define GnlAssocFormalPort(a)         (((a)->FormalPort))
#define GnlAssocActualPort(a)         (((a)->ActualPort))
#define GnlAssocHook(a)              (((a)->Hook))

#define SetGnlAssocFormalPort(a,f)   (((a)->FormalPort = f))
#define SetGnlAssocActualPort(a,f)   (((a)->ActualPort = f))
#define SetGnlAssocHook(a,f)         (((a)->Hook = f))

/* -----
/* GNL_USER_COMPONENT
/* -----
#define GNL_FORMAL_CHAR '0'
#define GNL_FORMAL_VAR  '1'
typedef struct GNL_USER_COMPONENT_STRUCT
{
```

gnl.h

```
/* This field must be the first of the structure ! */  
GNL_COMPONENT_TYPE Type;  
  
char *Name;  
char *InstanceName;  
char FormalType;  
/* Type of formal ports in the interface*/  
/* GNL_FORMAL_CHAR --> char *, or */  
/* GNL_FORMAL_VAR --> GNL_VAR */  
BLIST Interface; /* List of [GNL_ASSOC] */  
void *GnlDef; /* Gnl to which it refers. */  
void *CellDef; /* Lib Cell to which it refers. */  
int LineNumber;  
void *Hook;  
}  
GNL_USER_COMPONENT_REC, *GNL_USER_COMPONENT ;  
  
#define GnlUserComponentType(c) (((c)->Type))  
#define GnlUserComponentName(c) (((c)->Name))  
#define GnlUserComponentInstName(c) (((c)->InstanceName))  
#define GnlUserComponentFormalType(c) (((c)->FormalType))  
#define GnlUserComponentInterface(c) (((c)->Interface))  
#define GnlUserComponentGnlDef(c) (((c)->GnlDef))  
#define GnlUserComponentCellDef(c) (((c)->CellDef))  
#define GnlUserComponentLineNumber(c) (((c)->LineNumber))  
#define GnlUserComponentHook(c) (((c)->Hook))  
#define GnlUserComponentEquivCells(c) (((c)->Hook))  
  
#define SetGnlUserComponentType(c,n) (((c)->Type = n))  
#define SetGnlUserComponentName(c,n) (((c)->Name = n))  
#define SetGnlUserComponentInstName(c,n) (((c)->InstanceName = n))  
#define SetGnlUserComponentFormalType(c,n) (((c)->FormalType = n))  
#define SetGnlUserComponentInterface(c,n) (((c)->Interface = n))  
#define SetGnlUserComponentGnlDef(c,n) (((c)->GnlDef = n))  
#define SetGnlUserComponentCellDef(c,n) (((c)->CellDef = n))  
#define SetGnlUserComponentLineNumber(c,n) (((c)->LineNumber = n))  
#define SetGnlUserComponentHook(c,n) (((c)->Hook = n))  
#define SetGnlUserComponentEquivCells(c,n) (((c)->Hook = n))  
  
/* ----- */  
/* GNL_TRISTATE_COMPONENT */  
/* ----- */  
typedef struct GNL_TRISTATE_COMPONENT_STRUCT  
{  
/* This field must be the first of the structure ! */  
GNL_COMPONENT_TYPE Type;  
  
char *InstanceName;  
BLIST Interface; /* [Input,Output,Select] */  
int InputPol;  
int SelectPol;  
void *Hook;  
}  
GNL_TRISTATE_COMPONENT_REC, *GNL_TRISTATE_COMPONENT;  
  
#define GnlGetAssoc(c,i)\  
((GNL_ASSOC)BListElt ((c)->Interface, i))
```

gnl.h

```
#define GnlAssocActualPort(a)          (((a)->ActualPort))

#define GnlTriStateType(c)             (((c)->Type))
#define GnlTriStateInstName(c)        (((c)->InstanceName))
#define GnlTriStateInterface(c)       (((c)->Interface))
#define GnlTriStateInputAssoc(c)      (GnlGetAssoc(c,0))
#define GnlTriStateInput(c)           \
    GnlAssocActualPort(GnlTriStateInputAssoc(c))
#define GnlTriStateInputPol(c)        (((c)->InputPol))
#define GnlTriStateOutputAssoc(c)     (GnlGetAssoc(c,1))
#define GnlTriStateOutput(c)          \
    GnlAssocActualPort(GnlTriStateOutputAssoc(c))
#define GnlTriStateSelectAssoc(c)     (GnlGetAssoc(c,2))
#define GnlTriStateSelect(c)          \
    GnlAssocActualPort(GnlTriStateSelectAssoc(c))
#define GnlTriStateSelectPol(c)       (((c)->SelectPol))
#define GnlTriStateHook(c)            (((c)->Hook))

#define SetGnlTriStateType(c,n)        (((c)->Type = n))
#define SetGnlTriStateInstName(c,n)   (((c)->InstanceName = n))
#define SetGnlTriStateInterface(c,n)  (((c)->Interface = n))
#define SetGnlTriStateInput(c,n)      \
    SetGnlAssocActualPort(GnlGetAssoc(c,0),n)
#define SetGnlTriStateInputPol(c,n)   (((c)->InputPol = n))
#define SetGnlTriStateOutput(c,n)    \
    SetGnlAssocActualPort(GnlGetAssoc(c,1),n)
#define SetGnlTriStateSelect(c,n)    \
    SetGnlAssocActualPort(GnlGetAssoc(c,2),n)
#define SetGnlTriStateSelectPol(c,n)  (((c)->SelectPol = n))
#define SetGnlTriStateHook(c,n)       (((c)->Hook = n))

/* -----
/* GNL_SEQUENTIAL_COMPONENT                                */
/* ----- */
typedef struct GNL_SEQUENTIAL_COMPONENT_STRUCT
{
    /* This field must be the first of the structure !          */
    GNL_COMPONENT_TYPE      Type;

    GNL_SEQUENTIAL_OP      Op ;
    GNL_FORM_OUTPUT FormOutputs;    /* GNL_Q_QBAR, GNL_Q, GNL_QBAR      */
    char                  *InstanceName ;
    BLIST                 Interface; /* [Input, Output, OutputBar,      */
                                    /* Clock, Reset, Set]           */
    int                  ClockPol ;
    int                  ResetPol;
    int                  SetPol;
    void                 *Hook;
}
    GNL_SEQUENTIAL_COMPONENT_REC, *GNL_SEQUENTIAL_COMPONENT ;

#define GnlSequentialCompoType(pb)          (((pb)->Type))
#define GnlSequentialCompoOp(pb)            (((pb)->Op))
#define GnlSequentialCompoFormOutput(pb)   (((pb)->FormOutputs))
#define GnlSequentialCompoInstName(pb)     (((pb)->InstanceName))
#define GnlSequentialCompoInterface(pb)    (((pb)->Interface))
#define GnlSequentialCompoInputAssoc(pb)   (GnlGetAssoc(pb,0))
```

gnl.h

```
#define GnlSequentialCompoInput (pb)          \
    GnlAssocActualPort (GnlSequentialCompoInputAssoc (pb)) \
#define GnlSequentialCompoOutputAssoc (pb)  (GnlGetAssoc (pb,1)) \
#define GnlSequentialCompoOutput (pb)          \
    GnlAssocActualPort (GnlSequentialCompoOutputAssoc (pb)) \
#define GnlSequentialCompoOutputBarAssoc (pb)  (GnlGetAssoc (pb,2)) \
#define GnlSequentialCompoOutputBar (pb)          \
    GnlAssocActualPort (GnlSequentialCompoOutputBarAssoc (pb)) \
#define GnlSequentialCompoClockAssoc (pb)  (GnlGetAssoc (pb,3)) \
#define GnlSequentialCompoClock (pb)          \
    GnlAssocActualPort (GnlSequentialCompoClockAssoc (pb)) \
#define GnlSequentialCompoClockPol (pb)        (((pb)->ClockPol)) \
#define GnlSequentialCompoResetAssoc (pb)  (GnlGetAssoc (pb,4)) \
#define GnlSequentialCompoReset (pb)          \
    GnlAssocActualPort (GnlSequentialCompoResetAssoc (pb)) \
#define GnlSequentialCompoResetPol (pb)        (((pb)->ResetPol)) \
#define GnlSequentialCompoSetAssoc (pb)        (GnlGetAssoc (pb,5)) \
#define GnlSequentialCompoSet (pb)          \
    GnlAssocActualPort (GnlSequentialCompoSetAssoc (pb)) \
#define GnlSequentialCompoSetPol (pb)        (((pb)->SetPol)) \
#define GnlSequentialCompoHook (pb)          (((pb)->Hook)) \
#define SetGnlSequentialCompoType (pb,o)      (((pb)->Type = o)) \
#define SetGnlSequentialCompoOp (pb,o)        (((pb)->Op = o)) \
#define SetGnlSequentialCompoFormOutput (pb,o) (((pb)->FormOutputs = o)) \
#define SetGnlSequentialCompoInstName (pb,n)  (((pb)->InstanceName = n)) \
#define SetGnlSequentialCompoInterface (pb,n)  (((pb)->Interface = n)) \
#define SetGnlSequentialCompoInput (pb,i)       \
    SetGnlAssocActualPort (GnlGetAssoc (pb,0),i) \
#define SetGnlSequentialCompoOutput (pb,o)     \
    SetGnlAssocActualPort (GnlGetAssoc (pb,1),o) \
#define SetGnlSequentialCompoOutputBar (pb,o)   \
    SetGnlAssocActualPort (GnlGetAssoc (pb,2),o) \
#define SetGnlSequentialCompoClock (pb,c)     \
    SetGnlAssocActualPort (GnlGetAssoc (pb,3),c) \
#define SetGnlSequentialCompoClockPol (pb,ct)  (((pb)->ClockPol = ct)) \
#define SetGnlSequentialCompoReset (pb,r)     \
    SetGnlAssocActualPort (GnlGetAssoc (pb,4),r) \
#define SetGnlSequentialCompoResetPol (pb,r)  (((pb)->ResetPol = r)) \
#define SetGnlSequentialCompoSet (pb,s)       \
    SetGnlAssocActualPort (GnlGetAssoc (pb,5),s) \
#define SetGnlSequentialCompoSetPol (pb,s)  (((pb)->SetPol = s)) \
#define SetGnlSequentialCompoHook (pb,h)     (((pb)->Hook = h)) \
/* ----- */ \
/* GNL_BIDIR_COMPONENT */ \
/* ----- */ \
typedef struct GNL_BIDIR_COMPONENT_STRUCT \
{ \
    /* This field must be the first of the structure ! */ \
    GNL_COMPONENT_TYPE Type; \
    \
    GNL_VAR Input; \
    GNL_VAR Output; \
} GNL_BIDIR_COMPONENT_REC, *GNL_BIDIR_COMPONENT;
```

gnl.h

```
#define GnlBidirType(c)          (((c)->Type))
#define GnlBidirInput(c)          (((c)->Input))
#define GnlBidirOutput(c)         (((c)->Output))

#define SetGnlBidirType(c,n)      (((c)->Type = n))
#define SetGnlBidirInput(c,n)     (((c)->Input = n))
#define SetGnlBidirOutput(c,n)    (((c)->Output = n))

/* -----
/* GNL_BUF_COMPONENT
/* -----
typedef struct GNL_BUF_COMPONENT_STRUCT
{
    /* This field must be the first of the structure !
    GNL_COMPONENT_TYPE Type;

    char                *InstanceName;
    BLIST               Interface;
    void                *Hook;
}
    GNL_BUF_COMPONENT_REC, *GNL_BUF_COMPONENT;

#define GnlBufType(c)            (((c)->Type))
#define GnlBufInstName(c)        (((c)->InstanceName))
#define GnlBufInterface(c)       (((c)->Interface))
#define GnlBufInputAssoc(c)     (GnlGetAssoc(c,0))
#define GnlBufInput(c)           \
                                GnlAssocActualPort(GnlBufInputAssoc(c))
#define GnlBufOutputAssoc(c)    (GnlGetAssoc(c,1))
#define GnlBufOutput(c)          \
                                GnlAssocActualPort(GnlBufOutputAssoc(c))
#define GnlBufHook(c)           (((c)->Hook))

#define SetGnlBufType(c,n)       (((c)->Type = n))
#define SetGnlBufInstName(c,n)   (((c)->InstanceName = n))
#define SetGnlBufInterface(c,n)  (((c)->Interface = n))
#define SetGnlBufInput(c,n)      \
                                SetGnlAssocActualPort(GnlBufInputAssoc(c),n)
#define SetGnlBufOutput(c,n)     \
                                SetGnlAssocActualPort(GnlBufOutputAssoc(c),n)
#define SetGnlBufHook(c,n)       (((c)->Hook = n))

/* -----
/* GNL_COMPONENT
/* -----
typedef struct GNL_COMPONENT_STRUCT
{
    GNL_COMPONENT_TYPE Type;           /* GNL_SEQUENTIAL_COMPO, */
    /* GNL_TRISTATE_COMPO,           */
    /* GNL_BUF_COMPO,               */
    /* GNL_MACRO_COMPO,             */
    /* GNL_USER_COMPO,              */
    /* GNL_BIDIR_COMPO             */
}
    GNL_COMPONENT_REC, *GNL_COMPONENT ;

#define GnlComponentType(c)         (((c)->Type))
```

gnl.h

```
#define SetGnlComponentType(c, t)      (((c)->Type = t))

/* -----
/* GNL_CLOCK_VAR
/* -----
typedef struct GNL_CLOCK_VAR_STRUCT
{
    GNL_VAR Clock;
    BLIST ListComponents; /* List of GNL_SEQUENTIAL_COMPONENT */
    void *Hook;
}
GNL_CLOCK_VAR_REC, *GNL_CLOCK_VAR;

#define GnlClockVarClock(c)          (((c)->Clock))
#define GnlClockVarComponents(c)     (((c)->ListComponents))
#define GnlClockVarHook(c)          (((c)->Hook))

#define SetGnlClockVarClock(c, cl)   (((c)->Clock = cl))
#define SetGnlClockVarComponents(c, b) (((c)->ListComponents = b))
#define SetGnlClockVarHook(c, b)     (((c)->Hook = b))

/* -----
/* GNL_PATH_COMPONENT
/* -----
typedef struct GNL_PATH_COMPONENT_STRUCT
{
    GNL_USER_COMPONENT Component;
    struct GNL_PATH_COMPONENT_STRUCT *Previous;
}
GNL_PATH_COMPONENT_REC, *GNL_PATH_COMPONENT;

#define GnlPathComponentComponent(c) ((c)->Component)
#define GnlPathComponentPrevious(c) ((c)->Previous)

#define SetGnlPathComponentComponent(c,w) ((c)->Component = w)
#define SetGnlPathComponentPrevious(c,w) ((c)->Previous = w)

/* -----
/* GNL_STRUCT
/* -----
typedef struct GNL_STRUCT
{
    int UniqueId;
    int Tag;
    intRefCount; /* Reference counter */
    GNL_TYPE Type;
    char *Name;
    char *SourceFileName;
    BLIST ListSourceFiles;
    unsigned Status;
    BLIST ListPorts;
    BLIST HasTableNames; /* Hash Tables of GNL_VAR */
    BLIST HasTableCompoNames; /* Hash Tables of names of */
                               /* user components */
    int InstanceId;
```

gnl.h

```
/* GNL_NODE management */  
BLIST NodesSegments; /* Segments of GNL_NODE */  
GNL_NODE FirstNode;  
GNL_NODE LastNode;  
BLIST ListFreeNodes;  
  
int NbIn;  
int NbOut;  
int NbLocal;  
BLIST InputVars; /* List of GNL_VAR */  
BLIST OutputVars; /* List of GNL_VAR */  
BLIST LocalVars; /* List of GNL_VAR */  
BLIST Functions; /* List of GNL_VAR */  
BLIST ListClocks; /* List of GNL_CLOCK_VAR */  
BLIST ListComponents; /* Sequential, user and macros */  
/* components. */  
int NbDffs;  
int NbLatches;  
int NbTristates;  
int NbBuffers;  
int NbLitt;  
int LineNumber;  
  
/* List of Paths of Component which uses the current Gnl. Pointed */  
/* objects are of type [BLIST]. */  
/* This is a Hash List of [GNL_PATH_COMPONENT] */  
BLIST ListPathComponent;  
  
/* fields to store data information. */  
double Area;  
double NetArea;  
int Depth;  
int MaxFanout;  
float TechMax;  
float TechMin;  
  
/* fields to store FSM-input related information. */  
BLIST ListStateVar; /* List of GNL_VAR */  
BLIST HashListFsmVar; /* Hash list of fsm vars. */  
  
struct GNL_STRUCT *SynthesizedGnl;  
void *Hook ;  
}  
GNL_REC, *GNL;  
  
#define GnlUniqueId(g) (((g)->UniqueId))  
#define GnlTag(g) (((g)->Tag))  
#define GnlRefCount(g) (((g)->RefCount))  
#define GnlType(g) (((g)->Type))  
#define GnlName(g) (((g)->Name))  
#define GnlSourceFileName(g) (((g)->SourceFileName))  
#define GnlListSourceFiles(g) (((g)->ListSourceFiles))  
#define GnlStatus(g) (((g)->Status))  
#define GnlListPorts(g) (((g)->ListPorts))  
#define GnlHashNames(g) (((g)->HasTableNames))  
#define GnlHashCompoNames(g) (((g)->HasTableCompoNames))
```

gnl.h

```
#define GnlInstanceId(g)          (((g)->InstanceId))
#define GnlNodesSegments(g)        (((g)->NodesSegments))
#define GnlFirstNode(g)           (((g)->FirstNode))
#define GnlLastNode(g)            (((g)->LastNode))
#define GnlListFreeNodes(g)       (((g)->ListFreeNodes))
#define GnlNbIn(g)                (((g)->NbIn))
#define GnlNbOut(g)               (((g)->NbOut))
#define GnlNbLocal(g)             (((g)->NbLocal))
#define GnlInputs(g)              (((g)->InputVars))
#define GnlOutputs(g)             (((g)->OutputVars))
#define GnlLocals(g)              (((g)->LocalVars))
#define GnlFunctions(g)           (((g)->Functions))
#define GnlClocks(g)              (((g)->ListClocks))
#define GnlComponents(g)          (((g)->ListComponents))
#define GnlNbDffs(g)              (((g)->NbDffs))
#define GnlNbLatches(g)           (((g)->NbLatches))
#define GnlNbTristates(g)         (((g)->NbTristates))
#define GnlNbBuffers(g)           (((g)->NbBuffers))
#define GnlNbLitt(g)              (((g)->NbLitt))
#define GnlListPathComponent(g)   (((g)->ListPathComponent))
#define GnlLineNumber(g)          (((g)->LineNumber))
#define GnlArea(g)                (((g)->Area))
#define GnlNetArea(g)             (((g)->NetArea))
#define GnlDepth(g)               (((g)->Depth))
#define GnlMaxFanout(g)           (((g)->MaxFanout))
#define GnlTechMax(g)             (((g)->TechMax))
#define GnlTechMin(g)             (((g)->TechMin))
#define GnlListStateVar(g)        (((g)->ListStateVar))
#define GnlHashListFsmVar(g)      (((g)->HashListFsmVar))
#define GnlSynthesizedGnl(g)     (((g)->SynthesizedGnl))
#define GnlHook(g)                (((g)->Hook))

#define IncrGnlTag(g)             (((g)->Tag++))
#define SetGnlUniqueId(g,t)       (((g)->UniqueId = t))
#define SetGnlTag(g,t)            (((g)->Tag = t))
#define SetGnlRefCount(g,t)       (((g)->RefCount = t))
#define SetGnlType(g,t)           (((g)->Type = t))
#define SetGnlName(g,n)           (((g)->Name = n))
#define SetGnlSourceFileName(g,s)  (((g)->SourceFileName = s))
#define SetGnlListSourceFiles(g,s) (((g)->ListSourceFiles = s))
#define SetGnlStatus(g,s)         (((g)->Status = s))
#define SetGnlListPorts(g,h)       (((g)->ListPorts = h))
#define SetGnlHashNames(g,h)       (((g)->HasTableNames = h))
#define SetGnlHashCompoNames(g,h)  (((g)->HasTableCompoNames = h))
#define SetGnlInstanceId(g,h)      (((g)->InstanceId = h))
#define SetGnlNodesSegments(g,s)   (((g)->NodesSegments = s))
#define SetGnlFirstNode(g,n)       (((g)->FirstNode = n))
#define SetGnlLastNode(g,n)        (((g)->LastNode = n))
#define SetGnlListFreeNodes(g,n)   (((g)->ListFreeNodes = n))
#define SetGnlNbIn(g,nbi)          (((g)->NbIn = nbi))
#define SetGnlNbOut(g,nbo)         (((g)->NbOut = nbo))
#define SetGnlNbLocal(g,nbl)       (((g)->NbLocal = nbl))
#define SetGnlInputs(g,i)          (((g)->InputVars = i))
#define SetGnlOutputs(g,o)         (((g)->OutputVars = o))
#define SetGnlLocals(g,l)          (((g)->LocalVars = l))
#define SetGnlFunctions(g,f)       (((g)->Functions = f))
#define SetGnlClocks(g,c)          (((g)->ListClocks = c))
```

gnl.h

```

#define SetGnlComponents(g,w) (((g)->ListComponents = w))
#define SetGnlNbDffs(g,w) (((g)->NbDffs = w))
#define SetGnlNbLatches(g,w) (((g)->NbLatches = w))
#define SetGnlNbTristates(g,w) (((g)->NbTristates = w))
#define SetGnlNbBuffers(g,w) (((g)->NbBuffers = w))
#define SetGnlNbLitt(g,w) (((g)->NbLitt = w))
#define SetGnlListPathComponent(g,w) (((g)->ListPathComponent = w))
#define SetGnlLineNumber(g,w) (((g)->LineNumber = w))
#define SetGnlArea(g,w) (((g)->Area = w))
#define SetGnlNetArea(g,w) (((g)->NetArea = w))
#define SetGnlDepth(g,w) (((g)->Depth = w))
#define SetGnlMaxFanout(g,w) (((g)->MaxFanout = w))
#define SetGnlTechMax(g,w) (((g)->TechMax = w))
#define SetGnlTechMin(g,w) (((g)->TechMin = w))
#define SetGnlListStateVar(g,w) (((g)->ListStateVar = w))
#define SetGnlHashListFsmVar(g,w) (((g)->HashListFsmVar = w))
#define SetGnlSynthesizedGnl(g,w) (((g)->SynthesizedGnl = w))
#define SetGnlHook(g,h) (((g)->Hook = h))

/* -----
/* GNL_INFO_REC
/* ----- */
typedef struct GNL_INFO_STRUCT
{
    int NbInputs;
    int NbOutputs;
    int NbInOuts;
    int NbClocks;
    int NbFlipFlops;
    int NbLatches;
    int NbTriStates;
    int NbBuffers;
    int NbLiterals;
    float NbCombEquiGates;
    float NbSeqEquiGates;
    float NbTotalEquiGates;
    int MaxDepth;
}
GNL_INFO_REC, *GNL_INFO;

#define GnlInfoNbInputs(i) (((i)->NbInputs))
#define GnlInfoNbOutputs(i) (((i)->NbOutputs))
#define GnlInfoNbInOuts(i) (((i)->NbInOuts))
#define GnlInfoNbClocks(i) (((i)->NbClocks))
#define GnlInfoNbFlipFlops(i) (((i)->NbFlipFlops))
#define GnlInfoNbLatches(i) (((i)->NbLatches))
#define GnlInfoNbTriStates(i) (((i)->NbTriStates))
#define GnlInfoNbBuffers(i) (((i)->NbBuffers))
#define GnlInfoNbLit(i) (((i)->NbLiterals))
#define GnlInfoNbCombEquiGates(i) (((i)->NbCombEquiGates))
#define GnlInfoNbSeqEquiGates(i) (((i)->NbSeqEquiGates))
#define GnlInfoNbTotalEquiGates(i) (((i)->NbTotalEquiGates))
#define GnlInfoMaxDepth(i) (((i)->MaxDepth))

#define SetGnlInfoNbInputs(i,n) (((i)->NbInputs = n))
#define SetGnlInfoNbOutputs(i,n) (((i)->NbOutputs = n))
#define SetGnlInfoNbInOuts(i,n) (((i)->NbInOuts = n))

```

gnl.h

```
#define SetGnlInfoNbClocks(i,n)          (((i)->NbClocks = n))
#define SetGnlInfoNbFlipFlops(i,f)        (((i)->NbFlipFlops = f))
#define SetGnlInfoNbLatches(i,l)          (((i)->NbLatches = l))
#define SetGnlInfoNbTriStates(i,t)        (((i)->NbTriStates = t))
#define SetGnlInfoNbBuffers(i,t)          (((i)->NbBuffers = t))
#define SetGnlInfoNbLit(i,nb)            (((i)->NbLiterals = nb))
#define SetGnlInfoNbCombEquiGates(i,e)    (((i)->NbCombEquiGates = e))
#define SetGnlInfoNbSeqEquiGates(i,e)    (((i)->NbSeqEquiGates = e))
#define SetGnlInfoNbTotalEquiGates(i,e)   (((i)->NbTotalEquiGates = e))
#define SetGnlInfoMaxDepth(i,md)         (((i)->MaxDepth = md))

/* -----
/* GNL_NETWORK
/* -----
typedef struct GNL_NETWORK_STRUCT
{
    int             Tag;
    GNL           TopGnl;
    BLIST         ListGnls;

    double       Area;
    float        NLDelayArea;
    float        Period;
    int          Depth;
    int          NbDffs;
    int          NbLatches;
    int          NbTriStates;
    int          NbBuffers;

    void         *Hook;
}
GNL_NETWORK_REC, *GNL_NETWORK;

#define GnlNetworkTag(n)          (((n)->Tag))
#define GnlNetworkTopGnl(n)        (((n)->TopGnl))
#define GnlNetworkArea(n)          (((n)->Area))
#define GnlNetworkNLDelayArea(n)   (((n)->NLDelayArea))
#define GnlNetworkPeriod(n)        (((n)->Period))
#define GnlNetworkNbDffs(n)        (((n)->NbDffs))
#define GnlNetworkNbLatches(n)     (((n)->NbLatches))
#define GnlNetworkNbTriStates(n)   (((n)->NbTriStates))
#define GnlNetworkNbBuffers(n)     (((n)->NbBuffers))
#define GnlNetworkDepth(n)         (((n)->Depth))
#define GnlNetworkHook(n)          (((n)->Hook))

#define SetGnlNetworkTag(n,t)      (((n)->Tag = t))
#define SetGnlNetworkTopGnl(n,t)   (((n)->TopGnl = t))
#define SetGnlNetworkArea(n,t)     (((n)->Area = t))
#define SetGnlNetworkNLDelayArea(n,t)  (((n)->NLDelayArea = t))
#define SetGnlNetworkPeriod(n,t)   (((n)->Period = t))
#define SetGnlNetworkNbDffs(n,t)   (((n)->NbDffs = t))
#define SetGnlNetworkNbLatches(n,t) (((n)->NbLatches = t))
#define SetGnlNetworkNbTriStates(n,t)  (((n)->NbTriStates = t))
#define SetGnlNetworkNbBuffers(n,t) (((n)->NbBuffers = t))
#define SetGnlNetworkDepth(n,t)    (((n)->Depth = t))
#define SetGnlNetworkHook(n,t)     (((n)->Hook = t))
```

gnl.h

```
/* ----- EOF ----- */  
#endif
```

© 2002-2003 All Rights Reserved by the Author

gnllib.c

```
/*-----*/  
/*-----*/  
/*      File:          gnllib.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
#include "blist.h"  
#include "gnl.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
#include "gnloption.h"  
  
#include "blist.e"  
#include "bbdd.e"  
  
/*-----*/  
/*-----*/  
/* DEFINE */  
/*-----*/  
#define TRACE_COMBO_CELL /* if we want to view combinational cells */  
/* extracted from the synopsys library with */  
/* their boolean function. */  
#undef TRACE_COMBO_CELL  
  
#define GNL_MIN_NB_CANONICAL_ELEM 2000 /* !4 */  
#define GNL_MAX_NB_CANONICAL_ELEM 2000 /* !6 */  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern BDD_PTR GetBddPtrFromBdd ();  
extern Uint32 GetBddPtrRefCount ();  
extern GNL_ENV G_GnlEnv;  
  
/*-----*/  
/* GLOBAL VARIABLES */  
/*-----*/  
static GNL_VAR G_VAR_ZERO;  
static GNL_VAR G_VAR_ONE;  
static int G_MaxNbDeriveCells;  
  
/*-----*/  
/* GnlLibCreate */  
/*-----*/  
GNL_STATUS GnlLibCreate (GnlLib)  
    GNL_LIB *GnlLib;  
{  
  
    if ((*GnlLib = (GNL_LIB)malloc (1, sizeof (GNL_LIB_REC))) == NULL)  
        return (GNL_MEMORY_FULL);
```

gnllib.c

```
    return (GNL_OK);
}

/*-----*/
/* GnlLibDeriveCellCreate */
/*-----*/
GNL_STATUS GnlLibDeriveCellCreate (DeriveCell)
    LIB_DERIVE_CELL      *DeriveCell;
{

    if ((*DeriveCell = (LIB_DERIVE_CELL)
            calloc (1, sizeof (LIB_DERIVE_CELL_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlVarLibInfoCreate */
/*-----*/
GNL_STATUS GnlVarLibInfoCreate (VarLibInfo)
    GNL_VAR_LIB_INFO      *VarLibInfo;
{
    BLIST     NewList;

    if ((*VarLibInfo = (GNL_VAR_LIB_INFO)
            calloc (1, sizeof (GNL_VAR_LIB_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlVarLibInfoSymmetrics (*VarLibInfo, NewList);

    return (GNL_OK);
}

/*-----*/
/* GnlLibCellCreate */
/*-----*/
GNL_STATUS GnlLibCellCreate (LibCell)
    LIB_CELL *LibCell;
{
    if ((*LibCell = (LIB_CELL)calloc (1, sizeof (LIB_CELL_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlCreateInputData */
/*-----*/
GNL_STATUS GnlCreateInputData (InputData)
    LIB_INPUT_DATA *InputData;
{
    if ((*InputData = (LIB_INPUT_DATA)
            calloc (1, sizeof (LIB_INPUT_DATA_REC))) == NULL)
```

gnllib.c

```
    return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlSinglePinName
*/
/*-----*/
/* Verifies if the list of name has a single element or several ones */
/*-----*/
int GnlSinglePinName (ListPinName)
    LIBC_NAME_LIST ListPinName;
{

    if (!ListPinName)
        return (0);

    if (LibNameListNext(ListPinName))
        return (0);

    return (1);
}

/*-----
/* apply Unary_op
*/
/*-----*/
BDD_STATUS apply_unary_op (Mode, bdd_func, ListNodes, NewBdd)
    int          Mode;
    BDD_STATUS   (bdd_func) ();
    BLIST       ListNodes;
    BDD         *NewBdd;
{
    BDD_STATUS   Status;
    BDD         Bdd1;

    switch (Mode) {
        case 0:
            Bdd1 = GnlNodeBdd ((GNL_NODE) BListElt (ListNodes, 0));
            break;
        default:
            Bdd1 = GnlMapNodeInfoBdd ((GNL_NODE)
                                      BListElt (ListNodes, 0));
            break;
    }

    if ((Status = (bdd_func) (Bdd1, NewBdd)))
        return (Status);

    return (BDD_OK);
}

/*-----
/* CmpIndexBddNode
*/
/*-----*/
```

gnllib.c

```
static int CmpIndexBddNode (Node1, Node2)
    GNL_NODE *Node1;
    GNL_NODE *Node2;
{
    Uint32    Index1;
    Uint32    Index2;

    Index1 = GetBddPtrIndex (GetBddPtrFromBdd (GnlNodeBdd (*Node1)));
    Index2 = GetBddPtrIndex (GetBddPtrFromBdd (GnlNodeBdd (*Node2)));

    if (Index1 > Index2)
        return (-1);

    if (Index1 == Index2)
        return (0);

    return (1);
}

/*-----
/* CmpIndexBddMapNode
*/
static int CmpIndexBddMapNode (Node1, Node2)
    GNL_NODE *Node1;
    GNL_NODE *Node2;
{
    Uint32    Index1;
    Uint32    Index2;

    Index1 = GetBddPtrIndex (GetBddPtrFromBdd (
                                GnlMapNodeInfoBdd (*Node1)));
    Index2 = GetBddPtrIndex (GetBddPtrFromBdd (
                                GnlMapNodeInfoBdd (*Node2)));

    if (Index1 > Index2)
        return (-1);

    if (Index1 == Index2)
        return (0);

    return (1);
}
/*-----
/* SortSonsWithIndex
*/
/* We sort the sons of the operator in the decreasing order according to*/
/* the top index of their associated Bdds. This is to apply first of all*/
/* Bdds which should be at the bottom of the final Bdd. */
/*
static void SortSonsWithIndex (Mode, ListNodes)
    int          Mode;
    BLIST       ListNodes;
```

gnllib.c

```
{  
  
/* Not necessary to sort if less than 3 arguments. */  
if (BListSize (ListNodes) < 3)  
    return;  
  
if (Mode)  
    qsort(BListAddress (ListNodes), BListSize (ListNodes),  
          sizeof (GNL_NODE), CmpIndexBddMapNode);  
else  
    qsort(BListAddress (ListNodes), BListSize (ListNodes),  
          sizeof (GNL_NODE), CmpIndexBddNode);  
  
}  
  
/*-----*/  
/* apply_commut_op */  
/*-----*/  
/* For GNL_OR, GNL_AND, GNL_XOR and GNL_XNOR */  
/*-----*/  
BDD_STATUS apply_commut_op (Mode, bdd_func, ListNodes, NewBdd)  
{  
    int Mode;  
    BDD_STATUS (bdd_func) ();  
    BLIST ListNodes;  
    BDD *NewBdd;  
  
    int i;  
    BDD_STATUS Status;  
    BDD Bdd1;  
    BDD Bdd2;  
    BDD BddRes;  
    GNL_NODE NodeI;  
  
    /* We sort the Sons according to the top most variable Id of their */  
    /* associated Bdd so that we decrease the number of Apply calls. */  
    SortSonsWithIndex (Mode, ListNodes);  
  
    /* There is a "UseBdd (Bdd1)" because we free Bdd1 later on in the */  
    /* for loop. */  
    NodeI = (GNL_NODE)BListElt (ListNodes, 0);  
    switch (Mode) {  
        case 0:  
            Bdd1 = UseBdd (GnlNodeBdd (NodeI));  
            break;  
        default:  
            Bdd1 = UseBdd (GnlMapNodeInfoBdd (NodeI));  
            break;  
    }  
  
    for (i=1; i<BListSize (ListNodes); i++)  
    {  
        NodeI = (GNL_NODE)BListElt (ListNodes, i);  
    }
```

```

switch (Mode) {
    case 0:
        Bdd2 = GnlNodeBdd (NodeI);
        break;
    default:
        Bdd2 = GnlMapNodeInfoBdd (NodeI);
        break;
    }

if ((Status = (bdd_func) (Bdd1, Bdd2, &BddRes)))
    return (Status);

FreeBdd (Bdd1);

Bdd1 = BddRes;
}

*NewBdd = BddRes;

return (BDD_OK);
}

/*-----
/* apply_nand
*/
-----*/
BDD_STATUS apply_nand (Mode, ListNodes, NewBdd)
    int          Mode;
    BLIST      ListNodes;
    BDD         *NewBdd;
{
    int          i;
    BDD_STATUS   Status;
    GNL_NODE NodeI;
    BDD          BddRes;
    BDD          Bdd1;
    BDD          Bdd2;

/* We sort the Sons according to the top most variable Id of their */
/* associated Bdd so that we decrease the number of Apply calls. */
SortSonsWithIndex (Mode, ListNodes);

/* There is a "UseBdd (Bdd1)" because we free Bdd1 later on in the */
/* for loop. */
NodeI = (GNL_NODE)BListElt (ListNodes, 0);
switch (Mode) {
    case 0:
        Bdd1 = UseBdd (GnlNodeBdd (NodeI));
        break;
    default:
        Bdd1 = UseBdd (GnlMapNodeInfoBdd (NodeI));
        break;
}

```

gnllib.c

```
for (i=1; i<BListSize (ListNodes); i++)
{
    NodeI = (GNL_NODE)BListElt (ListNodes, i);

    switch (Mode) {
        case 0:
            Bdd2 = GnlNodeBdd (NodeI);
            break;
        default:
            Bdd2 = GnlMapNodeInfoBdd (NodeI);
            break;
    }

    /* each time we complement the previous result. */
    /* Z = NAND (a,b,c) -- NAND (a,b) = Y -- Z = NAND (!Y,c) */
    if ((Status = bdd_nand (ComplementBDD (Bdd1), Bdd2, &BddRes)))
        return (Status);

    FreeBdd (Bdd1);

    Bdd1 = BddRes;
}

*NewBdd = BddRes;

return (BDD_OK);
}

/*
 * apply_nor
 */
BDD_STATUS apply_nor (Mode, ListNodes, NewBdd)
    int          Mode;
    BLIST      ListNodes;
    BDD         *NewBdd;
{
    int          i;
    BDD_STATUS   Status;
    GNL_NODE NodeI;
    BDD          BddRes;
    BDD          Bdd1;
    BDD          Bdd2;

    /* We sort the Sons according to the top most variable Id of their */
    /* associated Bdd so that we decrease the number of Apply calls. */
    SortSonsWithIndex (Mode, ListNodes);

    /* There is a "UseBdd (Bdd1)" because we free Bdd1 later on in the */
    /* for loop. */
    NodeI = (GNL_NODE)BListElt (ListNodes, 0);
    switch (Mode) {
        case 0:
            Bdd1 = UseBdd (GnlNodeBdd (NodeI));
            break;
        default:
```

gnllib.c

```
Bdd1 = UseBdd (GnlMapNodeInfoBdd (NodeI));
break;
}

for (i=1; i<BListSize (ListNodes); i++)
{
    NodeI = (GNL_NODE)BListElt (ListNodes, i);

    switch (Mode) {
        case 0:
            Bdd2 = GnlNodeBdd (NodeI);
            break;
        default:
            Bdd2 = GnlMapNodeInfoBdd (NodeI);
            break;
    }

    /* each time we complement the previous result. */
/* Z = NOR (a,b,c) -- NOR (a,b) = Y -- Z = NOR (!Y,c)      */
    if ((Status = bdd_nor (ComplementBDD (Bdd1), Bdd2, &BddRes)))
        return (Status);

    FreeBdd (Bdd1);

    Bdd1 = BddRes;
}

*NewBdd = BddRes;

return (BDD_OK);
}

/*
 * GnlBddNodeApply
 */
BDD_STATUS GnlBddNodeApply (Mode, Gnl, Node, NewBdd)
    int          Mode;
    GNL          Gnl;
    GNL_NODE     Node;
    BDD          *NewBdd;
{
    BDD_STATUS     Status;
    BLIST         Sons;

    Sons = GnlNodeSons (Node);

    switch (GnlNodeOp (Node)) {
        case GNL_AND:
            if ((Status = apply_commut_op (Mode, bdd_and, Sons, NewBdd)))
                return (Status);
            break;

        case GNL_OR:
```

gnllib.c

```
if ((Status = apply_commut_op (Mode, bdd_or, Sons, NewBdd)))
    return (Status);
break;

case GNL_NAND:
    if ((Status = apply_nand (Mode, Sons, NewBdd)))
        return (Status);
break;

case GNL_NOR:
    if ((Status = apply_nor (Mode, Sons, NewBdd)))
        return (Status);
break;

case GNL_NOT:
    if ((Status = apply_unary_op (Mode, bdd_not, Sons, NewBdd)))
        return (Status);
break;

case GNL_XOR:
    if ((Status = apply_commut_op (Mode, bdd_xor, Sons, NewBdd)))
        return (Status);
break;

case GNL_XNOR:
    if ((Status = apply_commut_op (Mode, bdd_xnor, Sons, NewBdd)))
        return (Status);
break;

default:
    fprintf (stderr, "Operator not supported yet\n");
    exit (1);
}

return (BDD_OK);
}

/*-----
/* GnlCorrectNodeTag
*/
/*-----*/
/* tells if the node 'Node' has the same Tag as the general Gnl. */
/*-----*/
int GnlCorrectNodeTag (Gnl, Node)
    GNL          Gnl;
    GNL_NODE Node;
{
    return (GnlTag (Gnl) == GnlNodeTag (Node));
}

/*-----
/* GnlBuildBddOnNode
*/

```

gnllib.c

```
/*
 *-----*/
/* This procedure computes recursively the Bdd associated to GNL_NODE */
/* 'Node' and stores it in one of its fields (actually the Hook). */
/* On a reconvergent Node we re-use the Hook field as a previous */
/* computed Bdd only if the Tag of the Node and the Tag of the main Gnl */
/* are the same. */
/* This tag notion is usefull if one wants to recompute all the Bdds on */
/* tree: he just has to increment the Tag of the Gnl and everything is */
/* re-computed. */
/* Regarding the GNL_VARIABLE end case we take its corresponding Bdd */
/* thru 'GnlVarBdd' if this Var has no binded Var. If it has a binded */
/* var then we take the bdd of its binded Var. */
/* This is susefull to consider a same boolean tree under different var. */
/* ordering. */
/*-----*/
BDD_STATUS GnlBuildBddOnNode (Gnl, Node)
    GNL          Gnl;
    GNL_NODE Node;
{
    int           i;
    BDD_STATUS   Status;
    BLIST        Sons;
    GNL_NODE NodeI;
    BDD          NewBdd;
    GNL_VAR     Var;

    /* Bdd already computed for this node in a previous pass */
    if (GnlNodeBdd (Node) && GnlCorrectNodeTag (Gnl, Node))
    {
        UseBdd ((BDD)GnlNodeBdd (Node));
        return (BDD_OK);
    }

    SetGnlNodeTag (Node, GnlTag (Gnl));

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        if (GnlNodeSons (Node) == (BLIST)0)
            SetGnlNodeBdd (Node, bdd_zero());
        else
            SetGnlNodeBdd (Node, bdd_one());
        return (BDD_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        /* If the GNL_VAR 'Var' has a bind variable then we take its */
        /* corresponding Bdd. */
        /* This is useful to generate different ordering of the same */
        /* Boolean tree. */
        if (GnlVarBindVar (Var))
            Var = GnlVarBindVar (Var); /* we pick up the bind var */
        SetGnlNodeBdd (Node, UseBdd (GnlVarBdd (Var)));
        return (BDD_OK);
    }
}
```

gnllib.c

```
Sons = GnlNodeSons (Node);
for (i=0; i<BListSize (Sons); i++)
{
    NodeI = (GNL_NODE)BListElt (Sons, i);
    if ((Status = GnlBuildBddOnNode (Gnl, NodeI)))
        return (Status);
}

/* All the Bdd of the sons have been computed. Now we compute the Bdd*/
/* of Node. */
if ((Status = GnlBddNodeApply (0, Gnl, Node, &NewBdd)))
    return (Status);

/* We store the computed Bdd in the structure of 'Node'. */
SetGnlNodeBdd (Node, NewBdd);

return (BDD_OK);
}

/*
 * GnlSetBddInputs
 */
/*
 * Before invoking this function, a current BDD work space must exist.
 * (Use the function "InitBddWorkSpace" before).
 */
/*
 * This function creates an associate BDD for each Variable which is in
 * the list 'BList'. The ordering of Variables is the one in 'BList'.
 */
/*
 * Each Bdd of variable is stored thru 'SetGnlVarBdd'.
 */
BDD_STATUS GnlSetBddInputs (Gnl, BList)
    GNL          Gnl;
    BLIST        BList;
{
    int           i;
    BDD_VAR      VarBdd;
    BDD_STATUS   Status;
    GNL_VAR     Vari;

    for (i=0; i<BListSize (BList); i++)
    {
        Vari = (GNL_VAR)BListElt(BList, i);
        if ((Status = CreateBddVar (GnlVarName (Vari), &VarBdd)))
            return (Status);

        SetGnlVarBdd (Vari, BddVarBdd (VarBdd));
        SetGnlVarTag (Vari, GnlTag (Gnl));
    }

    return (BDD_OK);
}

/*
 * GnlGetBddMaxIndex
 */
/*
 * this procedure extracts the maximum of any variable in the Bdd 'Bdd'
 */
```

gnllib.c

```
/* Before calling this procedure you need to set '*MaxIndex' to 0 */
/*-----*/
static GnlGetBddMaxIndex (Bdd, MaxIndex)
    BDD          Bdd;
    Uint32       *MaxIndex;
{
    BDD_PTR   BddPtr;
    Uint32     Index;

    if (ConstantBdd (Bdd))
        return;

    BddPtr = GetBddPtrFromBdd (Bdd);
    Index = GetBddPtrIndex (BddPtr);

    if (Index > *MaxIndex)
        *MaxIndex = Index;

    GnlGetBddMaxIndex (GetBddPtrEdge1 (BddPtr), MaxIndex);
    GnlGetBddMaxIndex (GetBddPtrEdge0 (BddPtr), MaxIndex);
}

/*-----*/
/* LibBddInfoCreate
 */
/*-----*/
GNL_STATUS LibBddInfoCreate (BddPtr, NewBddInfo)
    BDD_PTR   BddPtr;
    LIB_BDD_INFO *NewBddInfo;
{
    BLIST     NewList;

    if ((*NewBddInfo =
        (LIB_BDD_INFO)malloc (1, sizeof (LIB_BDD_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetBddPtrHook (BddPtr, (int)*NewBddInfo);
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetLibBddInfoDeriveCells (BddPtr, NewList);

    return (GNL_OK);
}

/*-----*/
/* GnlBddAddBddLibInfo
 */
/*-----*/
/* This procedure creates a structure which is Hooked on each Bdd node. */
/* This structure is usefull to stored signature-type information */
/* Before calling this procedure all the Hook fields of each Bdd node */
/* must be NULL */
/*-----*/
GNL_STATUS GnlBddAddBddLibInfo (Bdd)
    BDD      Bdd;
{
    BDD_PTR   BddPtr;
```

```

LIB_BDD_INFO NewBddInfo;

BddPtr = GetBddPtrFromBdd (Bdd);

/* we already created a signature structure for this bdd node and      */
/* below ...                                                       */
if (GetBddPtrHook (BddPtr))
    return (GNL_OK);

if (LibBddInfoCreate (BddPtr, &NewBddInfo))
    return (GNL_MEMORY_FULL);

SetBddPtrHook (BddPtr, (int)NewBddInfo);

if (ConstantBdd (Bdd))
    return (GNL_OK);

if (GnlBddAddBddLibInfo (GetBddPtrEdge1 (BddPtr)))
    return (GNL_MEMORY_FULL);

if (GnlBddAddBddLibInfo (GetBddPtrEdge0 (BddPtr)))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----*/
/* GnlResetBddHook                                         */
/*-----*/
/* This procedure set all the Hook field of all the Bdd nodes to NULL. */
/* This function is not optimized (repass by already visited nodes) so */
/* it must be used on Bdd trees or small Bdds (like library functions) */
/* A bdd tag can be used to treat the reconvergent bdd node in oreder */
/* to optimize this but we do not know the value of each Tag here. So it*/
/* more robust to do the way below.                               */
/*-----*/
static void GnlResetBddHook (Bdd)
BDD Bdd;
{
    BDD_PTR BddPtr;

    if (ConstantBdd (Bdd))
    {
        BddPtr = GetBddPtrFromBdd (Bdd);
        SetBddPtrHook (BddPtr, (int)NULL);
        return;
    }

    BddPtr = GetBddPtrFromBdd (Bdd);
    SetBddPtrHook (BddPtr, (int)NULL);

    GnlResetBddHook (GetBddPtrEdge1 (BddPtr));
    GnlResetBddHook (GetBddPtrEdge0 (BddPtr));
}

```

gnllib.c

```
/*
 *-----*
 /* GnlComputeBddMinterms                               */
 /*-----*
 /* This procedure computes for each Bdd node the number of minterms of   */
 /* the On Set of the function when the Top bdd node variable is 0 or 1. */
 /* 'Min1' --> Nb minterm in the On Set when the Var = 1.                */
 /* 'Min0' --> Nb minterm in the On Set when the Var = 0.                  */
 /*-----*
 GNL_STATUS GnlComputeBddMinterms (Inv, Bdd, MaxIndex, Index, Min1, Min0)
 {
     int             Inv;
     BDD            Bdd;
     Uint32          MaxIndex;
     Uint32          *Index;
     int             *Min1;
     int             *Min0;
 {
     BDD_PTR        BddPtr;
     Uint32          Index0;
     Uint32          Index1;
     int             Min01;
     int             Min00;
     int             Min11;
     int             Min10;
     int             Power0;
     int             Power1;
     LIB_BDD_INFO    NewBddInfo;

     if (ConstantBdd (Bdd))
     {
         *Index = MaxIndex+1;
         if (((Bdd == bdd_one ()) && !Inv) ||
             ((Bdd == bdd_zero ()) && Inv))
         {
             *Min1 = 1;
             *Min0 = 0;
             return (GNL_OK);
         }
         *Min1 = 0;
         *Min0 = 0;
         return (GNL_OK);
     }

     BddPtr = GetBddPtrFromBdd (Bdd);
     *Index = GetBddPtrIndex (BddPtr);
     Inv ^= (Bdd != (BDD)BddPtr);

     if (GnlComputeBddMinterms (Inv, GetBddPtrEdge0 (BddPtr), MaxIndex,
                                &Index0, &Min01, &Min00))
         return (GNL_MEMORY_FULL);
     if (GnlComputeBddMinterms (Inv, GetBddPtrEdge1 (BddPtr), MaxIndex,
                                &Index1, &Min11, &Min10))
         return (GNL_MEMORY_FULL);

     Power0 = (int)Index0-(int)*Index-1;
     Power1 = (int)Index1-(int)*Index-1;
     *Min0 = (Min01+Min00);
 }
```

gnllib.c

```
while (Power0--) *Min0 = 2 * (*Min0);
*Min1 = (Min11+Min10);
while (Power1--) *Min1 = 2 * (*Min1);

if (!GetBddPtrHook (BddPtr))
{
    if (LibBddInfoCreate (BddPtr, &NewBddInfo))
        return (GNL_MEMORY_FULL);
    SetBddPtrHook (BddPtr, (int)NewBddInfo);
}

SetLibBddInfoMin0 (BddPtr, *Min0);
SetLibBddInfoMin1 (BddPtr, *Min1);

return (GNL_OK);
}

/*-----
/* GnlGetVarSignature
*/
/*-----*/
/* This procedure returns the signature couple (Min0, Min10 of the */
/* Variable of index 'Index'. */
/* Min0 = Number of Minterms of the OnSet of the function (Bdd) when the */
/* Var of Index 'Index' is set to 0. */
/* Min1 = Number of Minterms of the OnSet of the function (Bdd) when the */
/* Var of Index 'Index' is set to 1. */
*/
static void GnlGetVarSignature (Inv, Bdd, MaxIndex, Index, PreviousIndex,
                               Min0, Min1)
{
    int      Inv;
    BDD     Bdd;
    int      MaxIndex;
    int      Index;
    int      PreviousIndex;
    int      *Min0;
    int      *Min1;

    BDD_PTR BddPtr;
    Uint32   IndexBdd;
    int      Power;
    int      Min00;
    int      Min01;
    int      Min10;
    int      Min11;
    int      Min;
    int      NextInv;

    if (ConstantBdd (Bdd))
    {
        if (((Bdd == bdd_one ()) && !Inv) ||
            ((Bdd == bdd_zero ()) && Inv))
        {
            Min = 1;
            Power = (int)MaxIndex-(int)PreviousIndex-1;
            while (Power--) Min = 2 * Min;
        }
    }
}
```

gnllib.c

```
        *Min0 += Min;
        *Min1 += Min;
    }
    return;
}

BddPtr = GetBddPtrFromBdd (Bdd);
IndexBdd = GetBddPtrIndex (BddPtr);

if (Index == IndexBdd)
{
    Min = LibBddInfoMin1 (BddPtr);
    Power = (int)IndexBdd-(int)PreviousIndex-1;
    while (Power--) Min = 2 * Min;
    *Min1 += Min;
    Min = LibBddInfoMin0 (BddPtr);
    Power = (int)IndexBdd-(int)PreviousIndex-1;
    while (Power--) Min = 2 * Min;
    *Min0 += Min;
    return;
}

if (IndexBdd > Index)
{
    Min = LibBddInfoMin0 (BddPtr);
    Min += LibBddInfoMin1 (BddPtr);

    Power = (int)IndexBdd-(int)PreviousIndex-2;
    while (Power--) Min = 2 * Min;
    *Min0 += Min;
    *Min1 += Min;
    return;
}

NextInv = Inv ^ (Bdd != (BDD)BddPtr);

/* Otherwise the 'IndexBdd' is less than the var 'Index' we are      */
/* looking for.                                                       */
Min00 = Min01 = Min10 = Min11 = 0;
GnlGetVarSignature (NextInv, GetBddPtrEdge0 (BddPtr), MaxIndex, Index,
                    IndexBdd, &Min00, &Min01);
GnlGetVarSignature (NextInv, GetBddPtrEdge1 (BddPtr), MaxIndex, Index,
                    IndexBdd, &Min10, &Min11);

Min = Min00+Min10;
Power = (int)IndexBdd-(int)PreviousIndex-1;
while (Power--) Min = 2 * Min;
*Min0 += Min;
Min = Min01+Min11;
Power = (int)IndexBdd-(int)PreviousIndex-1;
while (Power--) Min = 2 * Min;
*Min1 += Min;

}

/*
 * -----
 * GnlComputeVarsSignatureFromBdd
 */
```

gnllib.c

```
/*
 *-----*/
/* This procedure computes the signature couple (Min0, Min1) for each */
/* bdd node which are represented in the bdd 'Bdd'. */
/*-----*/
GNL_STATUS GnlComputeVarsSignatureFromBdd (LibGnl, Bdd)
    GNL          LibGnl;
    BDD          Bdd;
{
    BDD_PTR  BddPtr;
    int       Min1;
    int       Min0;
    Uint32   BddMaxIndex;
    Uint32   Index;
    int       i;
    BLIST    ListInputs;
    GNL_VAR  VarI;

    ListInputs = GnlInputs (LibGnl);

    BddMaxIndex = 0;
    GnlGetBddMaxIndex (Bdd, &BddMaxIndex);

#ifndef 0
    /* We reset to NULL all the bdd nodes in 'BddNode'. */          */
    GnlResetBddHook (Bdd);                                           */
#endif

    /* We add the LIB_BDD_INFO structure on each hook of each Bdd node */
    if (GnlBddAddBddLibInfo (Bdd))
        return (GNL_MEMORY_FULL);

    /* for each Bdd node we compute the number of minterms of the On set */
    /* of the function when the bdd node var. is 0 or 1. */           */
    if (GnlComputeBddMinterms (0, Bdd, BddMaxIndex, &Index, &Min1, &Min0))
        return (GNL_MEMORY_FULL);

    BddPtr = GetBddPtrFromBdd (Bdd);
    SetLibBddInfoMin1 (BddPtr, Min1);
    SetLibBddInfoMin0 (BddPtr, Min0);

    for (i=1; i<=BddMaxIndex; i++)
    {
        Min0 = Min1 = 0;

        GnlGetVarSignature (0, Bdd, BddMaxIndex, i, 0, &Min0, &Min1);

        VarI = (GNL_VAR)BListElt (ListInputs, i-1);
        SetGnlVarMin0 (VarI, Min0);
        SetGnlVarMin1 (VarI, Min1);
    }

    return (GNL_OK);
}

/*
 *-----*/
/* GnlCompatibeSignature */
/*-----*/
```

gnllib.c

```
/*-----*/
int GnlCompatibeSignature (Var1, Var2)
    GNL_VAR  Var1;
    GNL_VAR  Var2;
{

    if (GnlVarMin0 (Var2) > GnlVarMin0 (Var1))
        return (1);

    if (GnlVarMin0 (Var2) < GnlVarMin0 (Var1))
        return (0);

    if (GnlVarMin1 (Var2) >= GnlVarMin1 (Var1))
        return (1);

    return (0);
}

/*-----*/
/* CmpVarSignature */
/*-----*/
int CmpVarSignature (Var1, Var2)
    GNL_VAR  *Var1;
    GNL_VAR  *Var2;
{

    if (GnlVarMin0 (*Var2) > GnlVarMin0 (*Var1))
        return (1);

    if (GnlVarMin0 (*Var2) < GnlVarMin0 (*Var1))
        return (-1);

    return (0);
}

/*-----*/
/* GnlListVarCompatibleSignature */
/*-----*/
/* This function verifies if the Variables respects the decreasing
/* ordering of their signatures.
/*-----*/
int GnlListVarCompatibleSignature (CellInputs)
    BLIST    CellInputs;
{
    int             i;
    GNL_VAR  VarI;
    GNL_VAR  VarINext;

    for (i=0; i<BListSize (CellInputs)-1; i++)
    {
        VarI = GnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i));
        VarINext = GnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i+1));
        if (!GnlCompatibeSignature (VarINext, VarI))
            return (0);
    }
    return (1);
}
```

```

/*
 * GnlSymmetricVars
 */
/*
 * This function returns 1 if the two Vars 'Var1' and 'Var2' are   */
 * symmetric.                                                 */
/*
int GnlSymmetricVars (Var1, Var2)
    GNL_VAR Var1;
    GNL_VAR Var2;
{
    int             i;
    GNL_VAR VarI;

return (0);

for (i=0; i<BListSize (GnlVarSymmetrics (Var1)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlVarSymmetrics (Var1), i);
    if (VarI == Var2)
        return (1);
}

return (0);
}

/*
 * GnlSymmetricInputs
 */
/*
GNL_STATUS GnlSymmetricInputs (Inputs, CellInputs)
    BLIST     Inputs;
    BLIST     CellInputs;
{
    int             i;
    GNL_VAR VarI;
    GNL_VAR VarCellI;
    int             NbDiff;

NbDiff = 0;
for (i=0; i<BListSize (Inputs); i++)
{
    VarI = (GNL_VAR)BListElt (Inputs, i);
    VarCellI = GnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i));
    if (VarI != VarCellI)
    {
        if (BListAddElt (GnlVarSymmetrics (VarI), (int)VarCellI))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlVarSymmetrics (VarCellI), (int)VarI))
            return (GNL_MEMORY_FULL);
    }
}

return (GNL_OK);
}

/*

```

gnllib.c

```
/* GnlVarHasSymmetricVar */  
/*-----*/  
/* This procedure returns '*Symmetric' set to 1 if there exists a derive*/  
/* cell which realizes the same Bdd 'Bdd' and which have only two switch*/  
/* variable inputs compare to the binded vars of 'CellInputs'. This is a*/  
/* two symmetric variable case and we store the symmetry information */  
/* between these two vars in the field 'GnlVarSymmetrics' of these two */  
/* vars. This will allow to cope the exploration tree later on when we */  
/* want to switch those two vars. */  
/*-----*/  
GNL_STATUS GnlVarHasSymmetricVar (Bdd, MotherCell, CellInputs, Symmetric)  
    BDD          Bdd;  
    LIB_CELL     MotherCell;  
    BLIST        CellInputs;  
    int          *Symmetric;  
{  
    BDD_PTR      BddPtr;  
    BLIST        ListDerivedCells;  
    int          i;  
    LIB_DERIVE_CELL DeriveCellI;  
  
    BddPtr = GetBddPtrFromBdd (Bdd);  
  
    /* No hook associated yet for the Bdd */  
    if (!GetBddPtrHook (BddPtr))  
    {  
        *Symmetric = 0;  
        return (GNL_OK);  
    }  
  
    ListDerivedCells = LibBddInfoDeriveCells (BddPtr);  
  
    for (i=0; i<BLISTSize (ListDerivedCells); i++)  
    {  
        DeriveCellI = (LIB_DERIVE_CELL)BLISTElt (ListDerivedCells, i);  
  
        /* If there is a derived cell realizing the same Bdd ... */  
        /* for the same mother cell. */  
        /* Then it is not necessary to create a derive cell. Moreover we */  
        /* store the information between symmetric variables. */  
        if ((Bdd == LibDeriveCellBdd (DeriveCellI)) &&  
            (LibDeriveCellMotherCell (DeriveCellI) == MotherCell))  
        {  
            /* We consider 1 to 1 one input between 'CellInputs' and */  
            /* 'LibDeriveCellInputs (DeriveCellI)' as symmetric */  
            if (GnlSymmetricInputs (LibDeriveCellInputs (DeriveCellI),  
                                   CellInputs))  
                return (GNL_MEMORY_FULL);  
            *Symmetric = 1;  
            return (GNL_OK);  
        }  
    }  
  
    *Symmetric = 0;  
    return (GNL_OK);  
}
```

gnllib.c

```
/*-----*/
/* GnlSaveInputSupport */
/*-----*/
GNL_STATUS GnlSaveInputSupport (Support, NewSupport)
    BLIST Support;
    BLIST *NewSupport;
{
    int i;
    BLIST SupportNode;
    GNL_VAR VarI;
    GNL_VAR VarNewI;

    if (BListCreateWithSize (BListSize (Support), NewSupport))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (Support); i++)
    {
        VarI = (GNL_VAR)BListElt (Support, i);
        VarNewI = GnlVarBindVar (VarI);
        if (BListAddElt (*NewSupport, (int)VarNewI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlRestoreInputSupport */
/*-----*/
void GnlRestoreInputSupport (Support, SavedSupport)
    BLIST Support;
    BLIST SavedSupport;
{
    int i;
    GNL_VAR VarI;
    GNL_VAR VarNewI;

    for (i=0; i<BListSize (Support); i++)
    {
        VarI = (GNL_VAR)BListElt (Support, i);
        VarNewI = (GNL_VAR)BListElt (SavedSupport, i);
        SetGnlVarBindVar (VarI, VarNewI);
    }
}

/*-----*/
/* GnlSubstituteInputSupport */
/*-----*/
/* Ex: Support = (i0 i1 i2) and OrderedSupport = (i1 i2 i0) then we */
/* have to substitute in 'SupportNode' i1 by i0, i2 by i1 and i0 by i2 */
/* so we get: SupportNode = (i2 i0 i1) */
/*-----*/
void GnlSubstituteInputSupport (Support, OrderedSupport)
    BLIST Support;
```

gnllib.c

```
BLIST          OrderedSupport;
{
    int          i;
    int          j;
    BLIST          SupportNode;
    GNL_VAR        VarI;
    GNL_VAR        VarJ;
    GNL_VAR        VarNewI;

    for (i=0; i<BListSize (OrderedSupport); i++)
    {
        /* We will substitute 'VarI' by the i-th element of 'Support'*/
        VarI = (GNL_VAR)BListElt (OrderedSupport, i);
        if ((VarI != G_VAR_ZERO) &&
            (VarI != G_VAR_ONE))
        {
            for (j=0; j<BListSize (Support); j++)
            {
                VarJ = (GNL_VAR)BListElt (Support, j);
                if (VarI == VarJ)
                    break;
            }

            /* We substitute bind Var of j-th Var in 'SupportNode' by      */
            /* i-th var of 'CellInputs'.                                         */
            SetGnlVarBindVar ((GNL_VAR)BListElt (Support, j),
                               (GNL_VAR)BListElt (Support, i));
        }
    }

    /*-----
    /* GnlGenerateDerivedCellRec                                     */
    /*-----*/
    static int          G_NbGnlGenerateDerivedCellRecCall;
    static GNL          G_LibGnl;
    static LIB_CELL      G_Cell;
    static GNL_NODE      G_CellFunction;
    static BLIST          G_CellInputs;

    GNL_STATUS GnlGenerateDerivedCellRec (Index)
        int          Index;
    {
        int          i;
        GNL_VAR        VarAux;
        GNL_VAR        VarI;
        GNL_VAR        VarIndex;
        BDD          Bdd;
        BDD_PTR        BddPtr;
        LIB_BDD_INFO    NewBddInfo;
        BLIST          NewList;
        LIB_DERIVE_CELL  NewDeriveCell;
        int          Symmetric;
        BLIST          SaveSupport;
```

```

G_NbGnlGenerateDerivedCellRecCall++;

if (BListSize (LibHCellDeriveCells (G_Cell)) > G_MaxNbDeriveCells)
    return (GNL_OK);

if (Index < BListSize (G_CellInputs)-1)
{
    if (GnlGenerateDerivedCellRec (Index + 1))
        return (GNL_MEMORY_FULL);

    for (i=Index+1; i<BListSize (G_CellInputs); i++)
    {
        VarI = GnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, i));
        VarIndex = GnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs,
                                                      Index));
        if ((VarI != G_VAR_ZERO) && (VarI != G_VAR_ONE) &&
            (VarIndex != G_VAR_ZERO) && (VarIndex != G_VAR_ONE))
        {
            VarIndex = GnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs,
                                                          Index));
            /* We use the pseudo canonical ordering that we got and we*/
            /* cope the exploration tree if the order is not compatibl*/
            /* We cope also the tree if the two Vars are symmetric      */
            if (!GnlSymmetricVars (VarIndex, VarI))
            {
                SetGnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, i),
                                  VarIndex);
                SetGnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, Index),
                                  VarI);

                if (GnlGenerateDerivedCellRec (Index + 1))
                    return (GNL_MEMORY_FULL);

                SetGnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, Index),
                                  VarIndex);
                SetGnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, i),
                                  VarI);
            }
        }
    }

    return (GNL_OK);
}

/* Then Index is equal to BListSize (G_CellInputs). We are at the      */
/* end of the recursion.                                              */
/* We increment the Tag of the Gnl so we will recompute the Bdd for   */
/* 'G_CellFunction' and not use the Hook field.                         */
SetGnlTag (G_LibGnl, GnlTag (G_LibGnl)+1);

/* BUG fix (05/20/99)                                                 */
#ifndef BUG_FIX
    if (GnlSaveInputSupport (G_CellInputs, &SaveSupport))
        return (GNL_MEMORY_FULL);
#endif

/* we substitute the variables in order to respect the pseudo order */

```

```

GnlSubstituteInputSupport (G_CellInputs, SaveSupport);
#endif

/* we build the new Bdd on the Boolean tree 'G_CellFunction'          */
if (GnlBuildBddOnNode (G_LibGnl, G_CellFunction))
    return (GNL_MEMORY_FULL);
Bdd = GnlNodeBdd (G_CellFunction);

/* Actually this Bdd has been already computed and we are in a           */
/* symmetric case.                                                       */
if (GnlVarHasSymmetricVar (Bdd, G_Cell, G_CellInputs, &Symmetric))
    return (GNL_MEMORY_FULL);

/* BUG fix (05/20/99)                                                 */
#ifndef BUG_FIX
    GnlRestoreInputSupport (G_CellInputs, SaveSupport);
    BListQuickDelete (&SaveSupport);
#endif

if (Symmetric)
    return (GNL_OK);

BddPtr = GetBddPtrFromBdd (Bdd);

/* We create the Bdd info structure if the Bdd has none in order to   */
/* attach it usefull informations like the Derive Cell which realizes*/
/* it.                                                               */
if (!GetBddPtrHook (BddPtr))
{
    if (LibBddInfoCreate (BddPtr, &NewBddInfo))
        return (GNL_MEMORY_FULL);
}

/* We create a new derive Cell and store all relevant informations     */
/* related to it.                                                       */
if (GnlLibDeriveCellCreate (&NewDeriveCell))
    return (GNL_MEMORY_FULL);
SetLibDeriveCellMotherCell (NewDeriveCell, G_Cell);
SetLibDeriveCellBdd (NewDeriveCell, Bdd);

if (BListCreateWithSize (BListSize (G_CellInputs), &NewList))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (G_CellInputs); i++)
{
    VarI = GnlVarBindVar ((GNL_VAR)BListElt (G_CellInputs, i));
    if (BListAddElt (NewList, (int)VarI))
        return (GNL_MEMORY_FULL);
}
SetLibDeriveCellInputs (NewDeriveCell, NewList);

/* we store on the Bdd node the fact that the current derive cell      */
/* realizes it.                                                       */
if (BListAddElt (LibBddInfoDeriveCells (BddPtr), (int)NewDeriveCell))
    return (GNL_MEMORY_FULL);

/* Adding this derive cell in the field of the Mother Cell            */
if (BListAddElt (LibHCellDeriveCells (G_Cell), (int)NewDeriveCell))

```

```

        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlGenerateDerivedCell
/*-----*/
GNL_STATUS GnlGenerateDerivedCell (LibGnl, Cell, Init)
    GNL           LibGnl;
    LIB_CELL Cell;
    int          Init;
{
    int          i;
    GNL_VAR     VarI;
    BLIST      CellInputs;
    BLIST      NewList;

    if (BListCreate (&NewList))
        return (GNL_MEMORY_FULL);

    SetLibHCellDeriveCells (Cell, NewList);

    CellInputs = LibHCellInputs (Cell);

    if (Init)
    {
        for (i=0; i<BListSize (CellInputs); i++)
        {
            VarI = (GNL_VAR)BListElt (CellInputs, i);

            /* Initialization: we bind each Variable by itself. */
            SetGnlVarBindVar (VarI, VarI);

            /* If the list of symmetric vars already exists we reset it */
            /* because it corresponds to the symmetric vars of the previous */
            /* cell function. */
            BSize (GnlVarSymmetries (VarI)) = 0;
        }
    }

    /* setting global variables for recursive calls. */
    G_LibGnl = LibGnl;
    G_Cell = Cell;
    G_CellFunction = LibHCellFunction (Cell);
    G_CellInputs = CellInputs;

    G_NbGnlGenerateDerivedCellRecCall = 0;
    /* We generate different Derived cells corresponding to each Bdd */
    if (GnlGenerateDerivedCellRec (0))
        return (GNL_MEMORY_FULL);

    /*
    fprintf (stderr, "NB CALL = %d\n", G_NbGnlGenerateDerivedCellRecCall);
    */
}

```

gnllib.c

```
    return (GNL_OK);
}

/*-----
/* GnlDerivedCell
/*-----
/* This procedure analyze a mother cell and derives from it a list of   */
/* Derived cells which corresponds to different flavour of the same   */
/* function (but different Bdds).                                         */
/*-----*/
GNL_STATUS GnlDerivedCell (LibGnl, Cell, Init)
    GNL           LibGnl;
    LIB_CELL     Cell;
    int          Init;
{
    BDD           BddFunction;
    int          i;
    int          j;
    GNL_VAR      VarI;
    LIB_DERIVE_CELL DeriveCellI;
    GNL_VAR      VarJ;

    SetGnlTag (LibGnl, GnlTag (LibGnl)+1);

    /* We build the Bdd corresponding to the function of the Cell which */
    /* is a Boolean tree (GNL_NODE).                                         */
    if (GnlBuildBddOnNode (LibGnl, LibHCellFunction (Cell)))
        return (GNL_MEMORY_FULL);

    BddFunction = GnlNodeBdd (LibHCellFunction (Cell));
    if (GnlComputeVarsSignatureFromBdd (LibGnl, BddFunction))
        return (GNL_MEMORY_FULL);

#ifdef TRACE_SIGNATURE
    for (i=0; i<BListSize (LibHCellInputs (Cell)); i++)
    {
        VarI = (GNL_VAR)BListElt (LibHCellInputs (Cell), i);
        fprintf (stderr, " <%s> Min0=%d Min1=%d\n", GnlVarName (VarI),
                 GnlVarMin0 (VarI), GnlVarMin1 (VarI));
    }
#endif

    if (GnlGenerateDerivedCell (LibGnl, Cell, Init))
        return (GNL_MEMORY_FULL);

#ifdef TRACE_LIB
    for (i=0; i<BListSize (LibHCellDeriveCells (Cell)); i++)
    {
        DeriveCellI = (LIB_DERIVE_CELL)
            BListElt (LibHCellDeriveCells (Cell), i);
        printf ("CELL <%s>\n",
               LibHCellName (LibDeriveCellMotherCell (DeriveCellI)));
        bdd_print (LibHDeriveCellBdd (DeriveCellI));
        for (j=0; j<BListSize (LibHDeriveCellInputs (DeriveCellI)); j++)
        {

```

gnllib.c

```
    VarJ = (GNL_VAR)
        BListElt (LibHDeriveCellInputs (DeriveCellI), j);
    printf ("%s ", GnlName (GnlVarBindVar (VarJ)));
}
printf ("\n\n");
}
#endif

    return (GNL_OK);
}

/*-----
/* GnlDegenerateCellRec
*/
-----*/
GNL_STATUS GnlDegenerateCellRec (LibGnl, Cell, Index)
    GNL           LibGnl;
    LIB_CELL      Cell;
    int           Index;
{
    int             i;
    GNL_VAR        VarI;
    BLIST          CellInputs;

    if (BListSize (LibHCellDeriveCells (Cell)) > G_MaxNbDeriveCells)
        return (GNL_OK);

    CellInputs = LibHCellInputs (Cell);

    if (Index < BListSize (CellInputs)-1)
    {
        if (GnlDegenerateCellRec (LibGnl, Cell, Index + 1))
            return (GNL_MEMORY_FULL);

        for (i=Index+1; i<BListSize (CellInputs); i++)
        {
            VarI = GnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i));

            SetGnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i),
                G_VAR_ZERO);
            if (GnlDegenerateCellRec (LibGnl, Cell, Index + 1))
                return (GNL_MEMORY_FULL);

            SetGnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i),
                G_VAR_ONE);
            if (GnlDegenerateCellRec (LibGnl, Cell, Index + 1))
                return (GNL_MEMORY_FULL);

            SetGnlVarBindVar ((GNL_VAR)BListElt (CellInputs, i), VarI);
        }
        return (GNL_OK);
    }

    G_NbGnlGenerateDerivedCellRecCall = 0;
    if (GnlDerivedCell (LibGnl, Cell, 0))
        return (GNL_MEMORY_FULL);
}
```

gnllib.c

```
    return (GNL_OK);
}

/*-----
/* GnlDegenerCell
*/
-----*/
GNL_STATUS GnlDegenerCell (LibGnl, Cell)
    GNL           LibGnl;
    LIB_CELL Cell;
{
    BLIST      CellInputs;
    int         i;
    GNL_VAR   VarI;

    CellInputs = LibHCellInputs (Cell);
    for (i=0; i<BListSize (CellInputs); i++)
    {
        VarI = (GNL_VAR)BListElt (CellInputs, i);

        /* Initialization: we bind each Variable by itself.          */
        SetGnlVarBindVar (VarI, VarI);

        /* If the list of symmetric vars already exists we reset it   */
        /* because it corresponds to the symmetric vars of the previous */
        /* cell function.                                              */
        BSize (GnlVarSymmetrics (VarI)) = 0;
    }

    if (GnlDegenerateCellRec (LibGnl, Cell, 0))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlBuildDerivedCells
*/
-----*/
GNL_STATUS GnlBuildDerivedCells (GnlLib)
    LIBC_LIB     GnlLib;
{
    int         i;
    GNL           LibGnl;
    LIBC_CELL   CellI;
    LIB_CELL HCellI;
    GNL_LIB    HGnlLib;
    int         NbComboCells;
    int         Index;
    LIBC_PIN   OutputPin;

    HGnlLib = (GNL_LIB)LibHook (GnlLib);
    LibGnl = GnlHLibGnl (HGnlLib);

    /* we count first the number of combinational cells.          */
}
```

gnllib.c

```
NbComboCells = 0;
CellI = LibCells (Gnllib);
for (; CellI != NULL; CellI = LibCellNext (CellI))
{
    /* If a sequential cell or cell not to use */ 
    if (LibCellFFLatch (CellI) || LibCellDontUse (CellI))
        continue;

    /* if it is a 3 states cell we return. */ 
    if (LibCellWith3StatePin (CellI, &OutputPin))
        continue;

    HCellI = (LIB_CELL)LibCellHook (CellI);

    /* We derive only combinational cells. */ 
    if (LibHCellType (HCellI) != CELL_IS_COMBINATORIAL)
        continue;

    /* The cell has no function. */ 
    if (!LibHCellFunction (HCellI))
        continue;

    NbComboCells++;
}

CellI = LibCells (Gnllib);
Index = 0;
for (; CellI != NULL; CellI = LibCellNext (CellI))
{
    /* If a sequential cell or cell not to use */ 
    if (LibCellFFLatch (CellI) || LibCellDontUse (CellI))
        continue;

    /* if it is a 3 states cell we return. */ 
    if (LibCellWith3StatePin (CellI, &OutputPin))
        continue;

    HCellI = (LIB_CELL)LibCellHook (CellI);

    /* We derive only combinational cells. */ 
    if (LibHCellType (HCellI) != CELL_IS_COMBINATORIAL)
        continue;

    /* The cell has no function. */ 
    if (!LibHCellFunction (HCellI))
        continue;

    Index++;

    if (!GnlEnvLog ())
    {
        fprintf (stderr, "%c Building Combinational Cell [%d/%d]", 13,
                 Index, NbComboCells);
    }

    if (GnlDerivedCell (LibGnl, HCellI, 1))
        return (GNL_MEMORY_FULL);
```

gnllib.c

```
#ifdef USE_DEGENERATED_CELL
    if (GnlDegenereCell (LibGnl, HCellI))
        return (GNL_MEMORY_FULL);
#endif

/*
    fprintf (stderr, " GATE <%s>: NB DERIVED CELLS = %d\n",
            LibHCellName (HCellI),
            BListSize (LibHCellDeriveCells (HCellI)));
*/
}

if (!GnlEnvLog ())
{
    fprintf (stderr, "%c Building Combinational Cell [%d/%d]\n\n", 13,
            Index, NbComboCells);
}

return (GNL_OK);
}

/*-----*/
/* GnlPrintLibc */
/*-----*/
void GnlPrintLibc (GnlLib)
    LIBC_LIB      GnlLib;
{
    LIBC_CELL      CellI;
    LIBC_PIN      Pins;
    LIBC_NAME_LIST ListPinName;

    CellI = LibCells (GnlLib);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        fprintf (stderr, "%s (", LibCellName(CellI));
        Pins = LibCellPins (CellI);
        for (; Pins != NULL; Pins = LibPinNext (Pins))
        {
            if (LibPinNext (Pins) == NULL)
                break;

            ListPinName = LibPinName (Pins);
            if (!GnlSinglePinName (ListPinName))
            {
                fprintf (stderr, "??, ");
            }
            else
            {
                fprintf (stderr, "%s, ", LibNameListName (ListPinName));
            }
        }

        if (Pins)
        {
            ListPinName = LibPinName (Pins);
```

gnllib.c

```
if (!GnlSinglePinName (ListPinName))
{
    fprintf (stderr, "?");
}
else
{
    fprintf (stderr, "%s", LibNameListName (ListPinName));
}
}

fprintf (stderr, ")\n");

fprintf (stderr, "\n\n\n\n\n");
}

/*-----*/
/* GnlBuildLibBdd */
/*-----*/
/* This procedure analyzes each cell of the library and build its pseudo*/
/* canonical Bdd representation. All these Bdds are stocked in the Bdd */
/* work space 'Ws' which is the Bdd image of the netlist 'GnllibGnl' of */
/* 'Gnllib'. For each Bdd node we store eventually the list of cells on */
/* which it can be mapped on. */
/*-----*/
GNL_STATUS GnlBuildLibBdd (Gnllib)
    LIBC_LIB Gnllib;
{
    GNL             LibGnl;
    int             MaxFanIn;
    GNL_STATUS      Status;
    BDD_WS          Ws;
    GNL_LIB         HGnllib;

    /* We pick up the "virtual" associated netlist of the current lib. */
    HGnllib = (GNL_LIB)LibHook (Gnllib);
    LibGnl = GnlHLibGnl (HGnllib);

    MaxFanIn = BListSize (GnlInputs (LibGnl));

    /* Initializing the Bdd work space */
    if ((Status = InitBddWorkSpace (MaxFanIn+1, &Ws)))
        return (GNL_MEMORY_FULL);
    SetGnlHLibBddWorkSpace (HGnllib, Ws);

    /* Creating Bdd for each primary input. */
    if ((Status = GnlSetBddInputs (LibGnl, GnlInputs (LibGnl))))
        return (Status);

    /* Giving terminal Bdd for specific variables */
    SetGnlVarBdd (G_VAR_ZERO, bdd_zero ());
    SetGnlVarBdd (G_VAR_ONE, bdd_one ());

    /* Now we derive all the cells of the Library. */
}
```

gnllib.c

```
if (GnlBuildDerivedCells (GnlLib))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlNormalizeInputsInNode
 */
/*
 * This procedures substitutes each old Variable by the associated
 * normalized one.
 */
/* We use the Hook field of the old variables to point on their new
 * unify variable. So we substitute old variables by their Hook field
 * if this one is not NULL.
 */
static BLIST      G_ListInputs;
static int        G_Index;
GNL_STATUS GnlNormalizeInputsInNode (Gnl, Node)
    GNL          Gnl;
    GNL_NODE    Node;
{
    int            i;
    GNL_NODE    SonI;
    GNL_VAR     Var;
    BLIST       Sons;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        /* If already passed thru this var we take the inputs $i which
         * has been previously associated to it.
         */
        if (GnlVarTag (Var) == GnlTag (Gnl))
        {
            SetGnlNodeSons (Node, (BLIST)GnlVarHook (Var));
            return (GNL_OK);
        }
        SetGnlVarTag (Var, GnlTag (Gnl));

        /* we affect to this Var the first available normalized input. To*/
        /* do so, we use the Hook field. */
        SetGnlVarHook (Var, (void*)BListElt (G_ListInputs, G_Index++));
        SetGnlNodeSons (Node, (BLIST)GnlVarHook (Var));
        return (GNL_OK);
    }

    Sons = GnlNodeSons (Node);
    for (i=0; i<BListSize (Sons); i++)
    {
        SonI = (GNL_NODE)BListElt (Sons, i);
        if (GnlNormalizeInputsInNode (Gnl, SonI))
            return (GNL_MEMORY_FULL);
    }
}
```

gnllib.c

```
    return (GNL_OK);
}

/*-----
/* GnlGetGnlNodeSupportRec                               */
/*-----*/
GNL_STATUS GnlGetGnlNodeSupportRec (GnlNode, Support)
    GNL_NODE      GnlNode;
    BLIST        *Support;
{
    int             i;
    GNL_NODE  SonI;
    GNL_VAR   Var;

    if (GnlNodeOp (GnlNode) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (GnlNode) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (GnlNode);
        if (!BListMemberOfList (*Support, Var, IntIdentical))
            if (BListAddElt (*Support, (int)Var))
                return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlNodeSons (GnlNode)); i++)
    {
        SonI = (GNL_NODE) BlistElt (GnlNodeSons (GnlNode), i);
        if (GnlGetGnlNodeSupportRec (SonI, Support))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlGetGnlNodeSupport                                */
/*-----*/
/* Returns the support of the Boolean Tree 'GnlNode'. The returned list */
/* is 'Support' and is composed of GNL_VAR elements.                      */
/*-----*/
GNL_STATUS GnlGetGnlNodeSupport (GnlNode, Support)
    GNL_NODE GnlNode;
    BLIST    *Support;
{

    if (BListCreateWithSize (1, Support))
        return (GNL_MEMORY_FULL);

    if (GnlGetGnlNodeSupportRec (GnlNode, Support))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

```

/*
 *-----*
 /* GnlGetInputs
 *-----*
 /* This procedure scans recursively the expression 'Node' and builds the*/
 /* list of GNL_VAR which are the leaves. This list is 'Inputs'. Each var*/
 /* is present only one time.                                         */
 *-----*
 GNL_STATUS GnlGetInputs (Node, Inputs)
    GNL_NODE Node;
    BLIST   Inputs;
{
    int          i;
    GNL_VAR   Var;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        if (!BListMemberOfList (Inputs, Var, IntIdentical))
            if (BListAddElt (Inputs, (int)Var))
                return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlGetInputs (SonI, Inputs))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 *-----*
 /* GnlGetOutputPinFromPins
 *-----*
 /* This procedure scans a list of Pins and returns the first output or  */
 /* inout pin encountered.                                              */
 /* If there is no such pin then a NULL value is assigned to 'OutputPin'.*/
 *-----*
void GnlGetOutputPinFromPins (Pins, OutputPin)
    LIBC_PIN Pins;
    LIBC_PIN *OutputPin;
{
    for (;Pins != NULL; Pins = LibPinNext(Pins))
    {
        if ((LibPinDirection (Pins) == OUTPUT_E) ||
            (LibPinDirection (Pins) == INOUT_E))
        {
            *OutputPin = Pins;
}

```

gnllib.c

```
        return;
    }

    *OutputPin = NULL;
}

/*
 * GnlGetListOutputPinFromPins
 */
/*
 * This procedure scans a list of Pins and returns the list of outputs
 * or inout pin encountered.
 * If there is no such pin then a NULL value is assigned to 'OutputPin'.
 */
GNL_STATUS GnlGetListOutputPinFromPins (Pins, ListOutputPin)
    LIBC_PIN    Pins;
    BLIST       *ListOutputPin;
{

    if (BListCreateWithSize (1, ListOutputPin))
        return (GNL_MEMORY_FULL);

    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        if ((LibPinDirection (Pins) == OUTPUT_E) ||
            (LibPinDirection (Pins) == INOUT_E))
        {
            if (BListAddElt (*ListOutputPin, (int)Pins))
                return (GNL_MEMORY_FULL);
        }
    }

    return (GNL_OK);
}

/*
 * GnlOpFromBoolOprOp
 */
/*
 * Returns the GNL_OP operator corresponding to the libc_bool_type_E one
 */
GNL_OP GnlOpFromBoolOprOp (BoolOprOp)
    libc_bool_type_E    BoolOprOp;
{

    switch (BoolOprOp) {
        case OR_B:
            return (GNL_OR);
        case AND_B:
            return (GNL_AND);
    }
}

/*
 * GnlGetGnlNodeFromBoolOpr
 */

```

gnllib.c

```
/*
 * This procedure returns the GNL_NODE expression thru 'GnlNode' of the *
 * original tree expression of type LIBC_BOOL_OPR generated in the LIBC *
 * parsing. Indexsignal, buses are not considered and we exit(1). */
/* The GNL_NODE objects are created in the memory segment of 'LibGnl'. */
*/
GNL_STATUS GnlGetGnlNodeFromBoolOpr (LibGnl, BoolOpr, GnlNode, Error)
    GNL                      LibGnl;
    LIBC_BOOL_OPR  BoolOpr;
    GNL_NODE      *GnlNode;
    int           *Error;
{
    text_buffer          *VarName;
    GNL_STATUS           GnlStatus;
    GNL_VAR              GnlVar;
    GNL_NODE             GnlNodeLeft;
    GNL_NODE             GnlNodeRight;
    BLIST                NewList;

    *Error = 0;

    /* terminal bool opr. */
    switch (LibBoolOprType (BoolOpr)) {
        /* terminal case of a signal. */
        case ID_B:
            VarName = LibBoolOprIdName (BoolOpr);
            if ((GnlStatus = GnlVarCreateAndAddInHashTable (LibGnl,
                    VarName, &GnlVar)))
            {
                if (GnlStatus != GNL_VAR_EXISTS)
                    return (GNL_MEMORY_FULL);
            }
            if (GnlCreateNode (LibGnl, GNL_VARIABLE, GnlNode))
                return (GNL_MEMORY_FULL);
            SetGnlNodeSons (*GnlNode, (BLIST)GnlVar);
            break;

        case ZERO_B:
            if (GnlCreateNodeVss (LibGnl, GnlNode))
                return (GNL_MEMORY_FULL);
            break;

        case ONE_B:
            if (GnlCreateNodeVdd (LibGnl, GnlNode))
                return (GNL_MEMORY_FULL);
            break;

        /* case of binary Boolean operators */
        case XOR_B:
        case OR_B:
        case AND_B:
            if (GnlGetGnlNodeFromBoolOpr (LibGnl,
                LibBoolOprLeftSon (BoolOpr),
                &GnlNodeLeft, Error))
                return (GNL_MEMORY_FULL);
    }
}
```

gnllib.c

```
if (*Error)
    return (GNL_OK);

if (GnlGetGnlNodeFromBoolOpr (LibGnl,
                               LibBoolOprRightSon (BoolOpr),
                               &GnlNodeRight, Error))
    return (GNL_MEMORY_FULL);

if (*Error)
    return (GNL_OK);

if (LibBoolOprType (BoolOpr) == XOR_B)
{
    if (GnlCreateNodeXor (LibGnl, GnlNodeLeft, GnlNodeRight,
                          GnlNode, 0))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)GnlNodeLeft))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)GnlNodeRight))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (LibGnl,
                  GnlOpFromBoolOprOp (LibBoolOprType (BoolOpr)),
                  GnlNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*GnlNode, NewList);
break;

/* Unary Boolean operators. It son is the right one */
case NOT_B:
    if (GnlGetGnlNodeFromBoolOpr (LibGnl,
                                   LibBoolOprRightSon (BoolOpr),
                                   &GnlNodeRight, Error))
        return (GNL_MEMORY_FULL);

    if (*Error)
        return (GNL_OK);

    if (GnlCreateNodeNot (LibGnl, GnlNodeRight, GnlNode))
        return (GNL_MEMORY_FULL);
    break;

default:
    *Error = 1;
    return (GNL_OK);
}

return (GNL_OK);

/*
-----*/
```

gnllib.c

```
/* GnlGetCellInputsOrigin */  
/*-----*/  
/* For each combinatorial cell we build the GNL_NODE of its function and*/  
/* extracts the inputs composing this function. These inputs are stored */  
/* in the list 'SetLibCellInputsOrigin'. */  
/*-----*/  
GNL_STATUS GnlGetCellInputsOrigin (LibGnl, Cell)  
    GNL          LibGnl;  
    LIBC_CELL    Cell;  
{  
    GNL_NODE     Node;  
    BLIST        NewList;  
    LIBC_PIN     Pins;  
    LIBC_PIN     OutputPin;  
    LIBC_BOOL_OPR Function;  
    LIB_CELL     HCell;  
    LIBC_NAME_LIST ListPinName;  
    char         *PinName;  
    int          Error;  
  
    /* It is a sequential cell. */  
    if (LibCellFFLatch (Cell))  
        return (GNL_OK);  
  
    /* if it is a 3 states cell we return. */  
    if (LibCellWith3StatePin (Cell, &OutputPin))  
        return (GNL_OK);;  
  
    Pins = LibCellPins (Cell);  
  
    /* we extract the first output Pin. If no output pin then we return */  
    GnlGetOutputPinFromPins (Pins, &OutputPin);  
    if (!OutputPin)  
        return (GNL_OK);  
  
    Function = LibPinFunction (OutputPin);  
  
    ListPinName = LibPinName (OutputPin);  
  
    /* Since 'PinName' can be a bundle a bus or a pin group we need to */  
    /* that it is a single bit signal. */  
    /* if there are different names we do not take this cell into account*/  
    if (!GnlSinglePinName (ListPinName))  
        return (GNL_OK);  
  
    PinName = LibNameListName (ListPinName);  
  
    if (!Function)  
    {  
        /* WARNING: we need to print out a warning message. */  
        return (GNL_OK);  
    }  
  
    /* Creating the GNL_NODE expression from the BOOL_OPR expression */
```

gnllib.c

```
if (GnlGetGnlNodeFromBoolOpr (LibGnl, Function, &Node, &Error))
    return (GNL_MEMORY_FULL);

if (Error)
    return (GNL_OK);

HCell = (LIB_CELL) LibCellHook (Cell);

/* If there is no real functionality then we continue. */
if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
{
    SetLibHCellFunction (HCell, NULL);
    SetLibHCellInputsOrigin (HCell, NULL);
    SetLibHCellType (HCell, CELL_IS_EXOTIC);
    return (GNL_OK);
}

/* Adding the ouput name. */
SetLibHCellOutput (HCell, PinName);

/* Then the cell is a combinatorial one. */
SetLibHCellType (HCell, CELL_IS_COMBINATORIAL);

/* Setting the field function of the gnl cell 'HCell'. The function */
/* is a GNL_NODE inherited from the BOOL_OPR expression. */
SetLibHCellFunction (HCell, Node);

#ifndef TRACE_COMBO_CELL
fprintf (stderr, "Cell <%s>: %s = ", LibHCellName (HCell), PinName);
GnlPrintNodeRec (stderr, Node, 0);
fprintf (stderr, "\n");
#endif

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);

SetLibHCellInputsOrigin (HCell, NewList);

if (GnlGetInputs (Node, NewList))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
-----*/
/* GnlUnifyInputsInLibGnl */
/*-----*/
/* This procedures redefines the primary inputs in order to put all the */
/* cells boolean functions dependant on these primary inputs. If for */
/* instance the MaxFanIn of the library is 'n' then we will create 'n' */
/* new internal primary inputs which are: $i1, $i2, ..., $in. In each */
/* boolean tree we will replace the old primary inputs by those ones. */
/* Ex: (a+b+c) . (!a+!b) --> ($i1+$i2+$i3) . (!$i1.!$i2) */
/* So the 'GnlLibGnl' of 'GnlLib' has new inputs which will be used by */
/* all the cells functions (e.g. GNL_NODE). */
/*-----*/
```

gnllib.c

```
/*-----*/
GNL_STATUS GnlUnifyInputsInLibGnl (Gnllib)
    LIBC_LIB Gnllib;
{
    int             i;
    GNL           LibGnl;
    GNL_VAR       NewVar;
    BLIST         NewList;
    LIBC_CELL     CellI;
    GNL_VAR_LIB_INFO VarLibInfo;
    BLIST         Support;
    GNL_LIB       HGnllib;
    LIB_CELL      HCellI;

    HGnllib = (GNL_LIB)LibHook (Gnllib);
    LibGnl = GnlHLibGnl (HGnllib);

    /* Creating List inputs for each Cell and extracting the Max Fan In */
    SetGnlHLibMaxFanIn (HGnllib, 0);
    CellI = LibCells (Gnllib);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        if (GnlGetCellInputsOrigin (LibGnl, CellI))
            return (GNL_MEMORY_FULL);

        HCellI = (LIB_CELL)LibCellHook (CellI);
        if (BListSize (LibHCellInputsOrigin (HCellI)) >
            GnlHLibMaxFanIn (HGnllib))
            SetGnlHLibMaxFanIn (HGnllib,
                                BListSize (LibHCellInputsOrigin (HCellI)));
    }

    /* we create the new unify list of inputs used by all the cells */
    /* functions. */
    /* BListCreate (&NewList) */
    /* return (GNL_MEMORY_FULL); */
    for (i=0; i<GnlHLibMaxFanIn (HGnllib); i++)
    {
        if (GnlCreateUniqueVar (LibGnl, "$i", &NewVar))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (NewList, (int)NewVar))
            return (GNL_MEMORY_FULL);
        /* Adding extra information on the Hook fields of these Vars */
        if (GnlVarLibInfoCreate (&VarLibInfo))
            return (GNL_MEMORY_FULL);
        SetGnlVarHook (NewVar, VarLibInfo);
    }

    /* we create two particular Variables which are GNL_VAR_ZERO and */
    /* GNL_VAR_ONE usefull for the degenerate pass. */
    if (GnlCreateUniqueVar (LibGnl, "$ZERO", &G_VAR_ZERO))
        return (GNL_MEMORY_FULL);
    if (GnlVarLibInfoCreate (&VarLibInfo))
        return (GNL_MEMORY_FULL);
    SetGnlVarHook (G_VAR_ZERO, VarLibInfo);
}
```

gnllib.c

```
if (GnlCreateUniqueVar (LibGnl, "$ONE", &G_VAR_ONE))
    return (GNL_MEMORY_FULL);
if (GnlVarLibInfoCreate (&VarLibInfo))
    return (GNL_MEMORY_FULL);
SetGnlVarHook (G_VAR_ONE, VarLibInfo);

BListQuickDelete (&GnlInputs (LibGnl));
SetGnlInputs (LibGnl, NewList);
SetGnlNbIn (LibGnl, BlistSize (GnlInputs (LibGnl)));
G_ListInputs = GnlInputs (LibGnl);

CellI = LibCells (Gnllib);
for (; CellI != NULL; CellI = LibCellNext (CellI))
{
    G_Index = 0;

    /* We increment the Gnl Tag in order to re-compute data in the      */
    /* 'GnlFunctionOnSet'.                                              */
    SetGnlTag (LibGnl, GnlTag (LibGnl)+1);

    /* If it is not a combinational gate we continue                  */
    HCellI = (LIB_CELL)LibCellHook (CellI);
    if (!LibHCellFunction (HCellI))
        continue;

    if (GnlNormalizeInputsInNode (LibGnl, LibHCellFunction (HCellI)))
        return (GNL_MEMORY_FULL);

    if (GnlGetGnlNodeSupport (LibHCellFunction (HCellI), &Support))
        return (GNL_MEMORY_FULL);

    /* The new inputs of the Cell 'CellI' is the support function      */
    SetLibHCellInputs (HCellI, Support);
}

return (GNL_OK);
}

/*-----
/* GnlGetInverters
*/
/*-----
/* This procedure extracts the list of inverters in the library and      */
/* stores it in "GnlLibInverters (Gnllib)".                           */
/*-----*/
GNL_STATUS GnlGetInverters (Gnllib)
    LIBC_LIB      Gnllib;
{
    GNL           LibGnl;
    BLIST         ListInputs;
    GNL_VAR       Var;
    GNL_NODE     NodeVar;
    GNL_NODE     NotNodeVar;
    BDD           Bdd;
    BDD_PTR      BddPtr;
    GNL_LIB      HGnllib;
```

gnllib.c

```
HGnllib = (GNL_LIB)LibHook (Gnllib);
LibGnl = GnlHLibGnl (HGnllib);
ListInputs = GnlInputs (LibGnl);
Var = (GNL_VAR)BListElt (ListInputs, 0);

if (GnlCreateNodeForVar (LibGnl, Var, &NodeVar))
    return (GNL_MEMORY_FULL);

if (GnlCreateNodeNot (LibGnl, NodeVar, &NotNodeVar))
    return (GNL_MEMORY_FULL);

if (GnlBuildBddOnNode (LibGnl, NotNodeVar))
    return (GNL_MEMORY_FULL);
SetGnlVarBindVar (Var, NULL);

Bdd = GnlNodeBdd (NotNodeVar);

BddPtr = GetBddPtrFromBdd (Bdd);
if (!GetBddPtrHook (BddPtr) ||
    (BListSize (LibBddInfoDeriveCells (BddPtr)) == 0))
{
    fprintf (stderr, "ERROR: not inverters defined in the library\n");
    return (GNL_MEMORY_FULL);
}

SetGnlHLibInverters (HGnllib, LibBddInfoDeriveCells (BddPtr));

return (GNL_OK);
}

/*-----
/* GnlGetSizeOfBasicEquivalentGate
/*-----
/* This procedure extracts the size of the basic gates of the current
/* library (e.g Minimum of AND2) and stores it in one of the field of
/* library 'HGnllib'.
/*-----*/
GNL_STATUS GnlGetSizeOfBasicEquivalentGate (Gnllib)
    LIBC_LIB      Gnllib;
{
    GNL          LibGnl;
    BLIST        ListInputs;
    GNL_VAR      Var1;
    GNL_VAR      Var2;
    GNL_NODE     NodeVar1;
    GNL_NODE     NodeVar2;
    GNL_NODE     NodeAnd2;
    BDD          Bdd;
    BDD_PTR      BddPtr;
    GNL_LIB      HGnllib;
    BLIST        NewList;
    LIB_DERIVE_CELL CellI;
    LIB_CELL      MotherCellI;
    LIB_CELL      BestCell;
    float         Area;
    int           i;
```

```

HGnllib = (GNL_LIB)LibHook (Gnllib);
LibGnl = GnlHLibGnl (HGnllib);
ListInputs = GnlInputs (LibGnl);
Var1 = (GNL_VAR)BListElt (ListInputs, 0);
Var2 = (GNL_VAR)BListElt (ListInputs, 1);

if (GnlCreateNodeForVar (LibGnl, Var1, &NodeVar1))
    return (GNL_MEMORY_FULL);

if (GnlCreateNodeForVar (LibGnl, Var2, &NodeVar2))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (LibGnl, GNL_AND, &NodeAnd2))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)NodeVar1))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)NodeVar2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NodeAnd2, NewList);

if (GnlBuildBddOnNode (LibGnl, NodeAnd2))
    return (GNL_MEMORY_FULL);

SetGnlVarBindVar (Var1, NULL);
SetGnlVarBindVar (Var2, NULL);

Bdd = GnlNodeBdd (NodeAnd2);

BddPtr = GetBddPtrFromBdd (Bdd);

/* There is no AND in the netlist */  

if (!GetBddPtrHook (BddPtr) ||
    (BListSize (LibBddInfoDeriveCells (BddPtr)) == 0))
{
    return (GNL_OK);
}

Area = 1000000000.0;
for (i=0; i<BListSize (LibBddInfoDeriveCells (BddPtr)); i++)
{
    CellI = (LIB_DERIVE_CELL)BListElt (LibBddInfoDeriveCells (BddPtr),
                                         i);
    MotherCellI = LibDeriveCellMotherCell (CellI);
    if ((Bdd == LibDeriveCellBdd (CellI)) &&
        (LibHCellArea (MotherCellI) < Area))
    {
        Area = LibHCellArea (MotherCellI);
        BestCell = MotherCellI;
    }
}

/* If not found any area for basic gate then we give 1.0 as area */  


```

gnllib.c

```
if (Area == 1000000000.0)
    Area = 1.0;

SetGnlHLibAreaBasicEqGate (HGnlLib, (double)Area);

return (GNL_OK);
}

/*
 * GnlCreateFillLibcHookStructures
 */
/*
 * This procedure creates our own structures related to GNL_LIB,   */
/* LIB_CELL, ... and hooks them to the LIBC structures. At the same time*/
/* some fields are set especially for the LIB_CELL structure.          */
/*
GNL_STATUS GnlCreateFillLibcHookStructures (GnlLibc)
    LIBC_LIB      GnlLibc;
{
    LIBC_CELL      CellI;
    LIB_CELL       LibCell;
    GNL_LIB        GnlLib;
    char           *NewName;

    if (GnlLibCreate (&GnlLib))
        return (GNL_MEMORY_FULL);

    /* Adding our own structure library to the original one           */
    SetLibHook (GnlLibc, GnlLib);

    CellI = LibCells (GnlLibc);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        if (GnlLibCellCreate (&LibCell))
            return (GNL_MEMORY_FULL);

        /* We fill some particular important fields of the LIB_CELL      */
        /* structure.                                                 */
        if (GnlStrCopy (LibCellName (CellI), &NewName))
            return (GNL_MEMORY_FULL);
        SetLibHCellName (LibCell, NewName);
        SetLibHCellLibcCell (LibCell, CellI);
        SetLibHCellLib (LibCell, GnlLib);

        /* Hooking libc cell 'CellI' with the gnl cell 'LibCell'. */
        SetLibCellHook (CellI, LibCell);
    }

    return (GNL_OK);
}

/*
 * GnlCellIsBuffer
 */
*/
```

gnllib.c

```
/* This function returns 1 if the cell 'Cell' is a buffer and 0 */
/* otherwise. */
*-----*/
int GnlCellIsBuffer (Cell)
    LIBC_CELL      Cell;
{
    LIBC_PIN      Pins;
    LIBC_PIN      OutputPin;
    LIBC_BOOL_OPR Function;

    Pins = LibCellPins (Cell);
    if (!Pins)
        return (0);

    if (!LibPinNext (Pins))
        return (0);

    if (LibPinNext(LibPinNext (Pins)) != NULL)
        return (0);

    GnlGetOutputPinFromPins (Pins, &OutputPin);
    Function = LibPinFunction (OutputPin);

    if (!Function)
        return (0);

    if (LibBoolOprType (Function) != ID_B )
        return (0);

    return (1);
}

*-----*/
/* GnlExtractLibcBuffers */
*-----*/
GNL_STATUS GnlExtractLibcBuffers (Gnllibc, LibcBuffers)
    LIBC_LIB      Gnllibc;
    BLIST      *LibcBuffers;
{
    int          i;
    LIBC_CELL     CellI;

    if (BListCreate (LibcBuffers))
        return (GNL_MEMORY_FULL);

    CellI = LibCells (Gnllibc);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        if (LibCellDontTouch (CellI) || LibCellDontUse (CellI) ||
            LibCellPad (CellI) )
            continue;

        if (!GnlCellIsBuffer (CellI))
            continue;
```

```

        if (BListAddElt (*LibcBuffers, CellI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlComputeCellArea
 */
/*-----*/
/* Returns the AREA of the cell 'Cell'. Here we pick up simply the value
/* stored in 'LibCellArea (Cell)'. */
/*
float GnlComputeCellArea (Cell)
    LIBC_CELL      Cell;
{
    return (LibCellArea (Cell));
}

/*-----
/* GnlComputeBddTransitionProbability
 */
/*-----*/
/* Returns the transition for the Cell 'Cell' to pass from the output
/* value '0' to '1'. we use a bdd representant to compute the card-
/* nality of the On set and Off set.
/*
GNL_STATUS GnlComputeBddTransitionProbability (Bdd, Proba)
    BDD          Bdd;
    float       *Proba;
{
    int          NbOnSetMinterm;
    int          NbOffSetMinterm;
    int          NbTotalMinterm;
    int          TotalTransitions;
    int          ToggleTransitions;
    Uint32      BddMaxIndex;
    Uint32      Index;
    int          Min1;
    int          Min0;

    BddMaxIndex = 0;
    GnlGetBddMaxIndex (Bdd, &BddMaxIndex);

    if (GnlComputeBddMinterms (0, Bdd, BddMaxIndex, &Index, &Min1,
                               &Min0))
        return (GNL_MEMORY_FULL);

    /* The On set cardinality id the sum of 'Min1' and 'Min0'.
       */
    NbOnSetMinterm = Min1 + Min0;

    NbTotalMinterm = 1;
    while (BddMaxIndex--) NbTotalMinterm = 2*NbTotalMinterm;
}

```

gnllib.c

```
NbOffSetMinterm = NbTotalMinterm-NbOnSetMinterm;

TotalTransitions = NbTotalMinterm * NbTotalMinterm;
ToggleTransitions = NbOnSetMinterm * NbOffSetMinterm;

*Proba = (float)ToggleTransitions/(float)TotalTransitions;

return (GNL_OK);
}

/*-----
/* GnlComputeCellChargeProba */
/*-----
/* Returns the transition for the Cell 'Cell' to pass from the output */
/* value '0' to '1'. we use a bdd representant to compute the card- */
/* nality of the On set and Off set. */
/* If the cell has no representant then we take 0.5 as probability. */
/*-----*/
GNL_STATUS GnlComputeCellChargeProba (Gnllibc, Cell, Proba)
    LIBC_LIB Gnllibc;
    LIBC_CELL Cell;
    float *Proba;
{
    LIB_CELL LibCell;
    float LoadCapacitance;
    BLIST ListEquivCells;
    LIB_DERIVE_CELL DeriveCell;
    BDD Bdd;

    LibCell = (LIB_CELL)LibCellHook (Cell);
    ListEquivCells = LibHCellDeriveCells (LibCell);

    if (ListEquivCells)
    {
        DeriveCell = (LIB_DERIVE_CELL)BListElt (ListEquivCells, 0);
        Bdd = LibDeriveCellBdd (DeriveCell);

        /* we compute the probability of the gate to transit from output */
        /* 0 to 1. */
        if (GnlComputeBddTransitionProbability (Bdd, Proba))
            return (GNL_MEMORY_FULL);

    /*
        fprintf (stderr, " GATE : %s %f\n", LibHCellName (LibCell),
                Proba);
    */

        return (GNL_OK);
    }

    *Proba = 0.5;

    return (GNL_OK);
}
```

gnllib.c

```
/*-----*/
/* GnlGetInternalPower */
/*-----*/
float GnlGetInternalPower (GnlLibc, Cell)
    LIBC_LIB GnlLibc;
    LIBC_CELL     Cell;
{
    float      Power;

    Power = LibCellPower (Cell) + LibCellCellLeakagePower (Cell);
    return (Power);
}

/*-----*/
/* GnlComputeCellsCostFunctionsInLibc */
/*-----*/
/* This procedure computes different value features on the cells of the */
/* library. This values are stored in the field of each LIB_CELL. The   */
/* values are:                                */
/*     - Area                                */
/*     - Probability of charge (0 --> 1)      */
/*     - Internal Power                      */
/*-----*/
GNL_STATUS GnlComputeCellsCostFunctionsInLibc (GnlLibc)
    LIBC_LIB GnlLibc;
{
    LIBC_CELL           CellI;
    LIB_CELL            LibCell;
    GNL_LIB             GnlLib;
    float               Area;
    float               Proba;
    float               InternalPower;

    CellI = LibCells (GnlLibc);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        LibCell = (LIB_CELL)LibCellHook (CellI);

        /* Computing the AREA value of the cell 'CellI'.          */
        Area = GnlComputeCellArea (CellI);
        SetLibHCellArea (LibCell, Area);

        /* Computing the charge probability of the cell 'CellI'. */
        if (GnlComputeCellChargeProba (GnlLibc, CellI, &Proba))
            return (GNL_MEMORY_FULL);
        SetLibHCellChargeProba (LibCell, Proba);

        /* Computing the total internal power comsumption which is the */
        /* sum of the leakage power and internal power.                */
        InternalPower = GnlGetInternalPower (GnlLibc, CellI);
        SetLibHCellInternalPower (LibCell, InternalPower);
    }

    return (GNL_OK);
}
```

```

/*
 *-----*/
/* GnlBuildLib */ 
/*-----*/
/* This procedure hooks our own library structure on the LIBC library */
/* structure. It analyzes each cell and put them in the right places. */
/* For combinational cells, thier Boolean function is derived into a set */
/* of canonical Bdds representants useful for the technology mapping */
/* phase. */
/*-----*/
GNL_STATUS GnlBuildLib (GnlLibc)
    LIBC_LIB GnlLibc;
{
    GNL             LibGnl;
    GNL_LIB        HGNLlib;
    BLIST          LibcDffs;
    BLIST          LibcLatches;
    BLIST          Libc3States;
    BLIST          LibcBuffers;

    if (GnlEnvBuildLibForce() == GNL_BUILD_LIB_MIN)
        G_MaxNbDeriveCells = GNL_MIN_NB_CANONICAL_ELEM;
    else
        G_MaxNbDeriveCells = GNL_MAX_NB_CANONICAL_ELEM;

    /* We add hooked structures on 'GnlLibc' and the cells. */
    if (GnlCreateFillLibcHookStructures (GnlLibc))
        return (GNL_MEMORY_FULL);

    HGNLlib = (GNL_LIB)LibHook (GnlLibc);
    SetGnlHLibName (HGNLlib, LibName (GnlLibc));
    if (GnlCreate (LibName (GnlLibc), &LibGnl))
        return (GNL_MEMORY_FULL);
    SetGnlHLibGnl (HGNLlib, LibGnl);

    /* We create new inputs for all the cells functions. All these */
    /* functions will depend upon these new unify variables. */
    if (GnlUnifyInputsInLibGnl (GnlLibc))
        return (GNL_MEMORY_FULL);

    /* We build the library by deriving cells according to their Bdds */
    /* representations (for each new Bdd representant of a cell we have */
    /* a new derived cell). */
    if (GnlBuildLibBdd (GnlLibc))
        return (GNL_MEMORY_FULL);

    /* We extract the list of all the inverters in the library and store */
    /* them in a special place to have direct access to them. */
    if (GnlGetInverters (GnlLibc))
        return (GNL_MEMORY_FULL);

    /* we compute different cost functions for all the Comb. cells like */
    /* (AREA, POWER, TIMING). */
    if (GnlComputeCellsCostFunctionsInLibc (GnlLibc))
        return (GNL_MEMORY_FULL);

```

```

/* We extract the minimum area of a basic equivalent gate (AND2) and */
/* store its value.                                              */
if (GnlGetSizeOfBasicEquivalentGate (GnlLibc))
    return (GNL_MEMORY_FULL);

/* extraction of the flip-flops and latches in the libc library      */
if (GnlExtractLibcSequential (GnlLibc, &LibcDffs, &LibcLatches))
    return (GNL_MEMORY_FULL);

/* extraction of the tri-states in the libc library                  */
if (GnlExtractLibc3States (GnlLibc, &Libc3States))
    return (GNL_MEMORY_FULL);

/* extraction of the buffers in the libc library                      */
if (GnlExtractLibcBuffers (GnlLibc, &LibcBuffers))
    return (GNL_MEMORY_FULL);

SetGnlHLibCellsDffs (HGnllib, LibcDffs);
SetGnlHLibCellsLatches (HGnllib, LibcLatches);
SetGnlHLibCells3States (HGnllib, Libc3States);
SetGnlHLibCellsBuffers (HGnllib, LibcBuffers);

fprintf (stderr, "\n", BListSize (LibcDffs));
fprintf (stderr, " Library Feature:\n", BListSize (LibcDffs));
fprintf (stderr, "     o NB. DFFS      = %d\n", BListSize (LibcDffs));
fprintf (stderr, "     o NB. LATCHES   = %d\n", BListSize (LibcLatches));
fprintf (stderr, "     o NB. 3-STATES = %d\n", BListSize (Libc3States));
fprintf (stderr, "     o NB. BUFFERS  = %d\n", BListSize (LibcBuffers));

return (GNL_OK);
}
/*-----*/

```

gnllibc.h

```
/*-----*/
/*
 *      File:          gnllib.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 */
/*-----*/
#ifndef GNLLIB_H
#define GNLLIB_H

/*-----*/
/* Info stored on GNL_NODE thru Hook field */
/*-----*/
#define GnlNodeBdd(n)      (((BDD)((n)->Hook)))
#define SetGnlNodeBdd(n, b) (((n)->Hook = (void*)b))

/*-----*/
/* Info stored on GNL_VAR thru Hook field */
/*-----*/
typedef struct GNL_VAR_LIB_INFO_STRUCT {
    BDD           Bdd;
    int            Min0;
    int            Min1;
    GNL_VAR     BindVar;
    BLIST        Symetrics; /* List of symmetric vars */
} GNL_VAR_LIB_INFO_REC, *GNL_VAR_LIB_INFO;

#define GnlVarBdd(n)      (((GNL_VAR_LIB_INFO)((n)->Hook))->Bdd)
#define GnlVarMin0(n)     (((GNL_VAR_LIB_INFO)((n)->Hook))->Min0)
#define GnlVarMin1(n)     (((GNL_VAR_LIB_INFO)((n)->Hook))->Min1)
#define GnlVarBindVar(n)  (((GNL_VAR_LIB_INFO)((n)->Hook))->BindVar)
#define GnlVarSymmetrics(n) (((GNL_VAR_LIB_INFO)((n)->Hook))->Symetrics)

#define SetGnlVarBdd(n, b) (((GNL_VAR_LIB_INFO)((n)->Hook))->Bdd = b)
#define SetGnlVarMin0(n,m) (((GNL_VAR_LIB_INFO)((n)->Hook))->Min0 = m)
#define SetGnlVarMin1(n,m) (((GNL_VAR_LIB_INFO)((n)->Hook))->Min1 = m)
#define SetGnlVarBindVar(n, b) \
    (((GNL_VAR_LIB_INFO)((n)->Hook))->BindVar = b)
#define SetGnlVarSymmetrics(n, s) \
    (((GNL_VAR_LIB_INFO)((n)->Hook))->Symetrics = s)

#define SetGnlVarLibInfoSymmetrics(v,s) ((v)->Symetrics = s)

/* ----- */
/* Info stored on bdd Node thru Hook field */
/*-----*/
typedef struct LIB_BDD_INFO
{
    BLIST    DeriveCells;
    int      NbMinterm0;
    int      NbMinterm1;
}
```

```

LIB_BDD_INFO_REC, *LIB_BDD_INFO;

#define LibBddInfoDeriveCells(bptr) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->DeriveCells)
#define LibBddInfoMin0(bptr) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->NbMinterm0)
#define LibBddInfoMin1(bptr) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->NbMinterm1)

#define SetLibBddInfoDeriveCells(bptr, d) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->DeriveCells = d)
#define SetLibBddInfoMin0(bptr,n) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->NbMinterm0 = n)
#define SetLibBddInfoMin1(bptr,n) \
    (((LIB_BDD_INFO)GetBddPtrHook(bptr))->NbMinterm1 = n)

/* -----
/* LIB_CELL_TYPE
/* -----
*/
typedef enum {
    CELL_IS_COMBINATORIAL,
    CELL_IS_SEQUENTIAL,
    CELL_IS_EXOTIC
} LIB_CELL_TYPE;

/* -----
/* LIB_INPUT_DATA
/* -----
*/
typedef struct LIB_INPUT_DATA_STRUCT
{
    float           InputLoad;
    float           MaxLoad;
    float           RiseBlockDelay;
    float           RiseFanoutDelay;
    float           FallBlockDelay;
    float           FallFanoutDelay;
}
LIB_INPUT_DATA_REC, *LIB_INPUT_DATA;

#define LibInputLoad(c)           (((c)->InputLoad))
#define LibMaxLoad(c)            (((c)->MaxLoad))
#define LibRiseBlockDelay(c)     (((c)->RiseBlockDelay))
#define LibRiseFanoutDelay(c)   (((c)->RiseFanoutDelay))
#define LibFallBlockDelay(c)    (((c)->FallBlockDelay))
#define LibFallFanoutDelay(c)  (((c)->FallFanoutDelay))

#define SetLibInputLoad(c,l)      (((c)->InputLoad) = l)
#define SetLibMaxLoad(c,l)       (((c)->MaxLoad) = l)
#define SetLibRiseBlockDelay(c,l) (((c)->RiseBlockDelay) = l)
#define SetLibRiseFanoutDelay(c,l) (((c)->RiseFanoutDelay) = l)
#define SetLibFallBlockDelay(c,l) (((c)->FallBlockDelay) = l)
#define SetLibFallFanoutDelay(c,l) (((c)->FallFanoutDelay) = l)

/* -----
/* LIB_OUTPUT_DATA
/* -----
*/

```

gnllibc.h

```
typedef struct LIB_OUTPUT_DATA_STRUCT
{
    float           Impedance;
}
LIB_OUTPUT_DATA_REC, *LIB_OUTPUT_DATA;

#define LibImpedance(c)          (((c)->Impedance))
#define SetLibImpedance(c,l)     (((c)->Impedance) = l)

/* -----
/* Hooked structure related to the CELL group
/* ----- */

typedef struct LIB_CELL_STRUCT
{
    char             *Name;
    LIB_CELL_TYPE    Type;
    int              *Lib;
    char             *Output;
    BLIST            Inputs;
    BLIST            InputsOrigin;
    float            Area;
    float            Timing;
    float            ChargeProba;
    float            InternalPower;
    GNL_NODE         Function;
    BLIST            DeriveCells;
    LIBC_CELL        LibcCell;
    int              *Hook;
}
LIB_CELL_REC, *LIB_CELL;

#define LibHCellName(c)          ((c)->Name)
#define LibHCellLibcCell(c)       ((c)->LibcCell)
#define LibHCellType(c)          ((c)->Type)
#define LibHCellLib(c)           ((GNL_LIB)((c)->Lib))
#define LibHCellOutput(c)        ((c)->Output)
#define LibHCellInputs(c)        ((c)->Inputs)
#define LibHCellInputsOrigin(c)  ((c)->InputsOrigin)
#define LibHCellArea(c)          ((c)->Area)
#define LibHCellTiming(c)        ((c)->Timing)
#define LibHCellChargeProba(c)   ((c)->ChargeProba)
#define LibHCellInternalPower(c) ((c)->InternalPower)
#define LibHCellFunction(c)      ((c)->Function)
#define LibHCellDeriveCells(c)   ((c)->DeriveCells)
#define LibHCellHook(c)          ((c)->Hook)

#define SetLibHCellName(c, n)     ((c)->Name = n)
#define SetLibHCellLibcCell(c, n)  ((c)->LibcCell = n)
#define SetLibHCellType(c, n)     ((c)->Type = n)
#define SetLibHCellLib(c, n)      (((c)->Lib) = (int*)n)
#define SetLibHCellOutput(c, n)   ((c)->Output = n)
#define SetLibHCellInputs(c, n)   ((c)->Inputs = n)
#define SetLibHCellInputsOrigin(c, n) ((c)->InputsOrigin = n)
#define SetLibHCellArea(c, n)     ((c)->Area = n)
#define SetLibHCellTiming(c, n)   ((c)->Timing = n)
#define SetLibHCellChargeProba(c, n) ((c)->ChargeProba = n)
```

gnllibc.h

```

#define SetLibHCellInternalPower(c, n)      ((c)->InternalPower = n)
#define SetLibHCellFunction(c, n)           ((c)->Function = n)
#define SetLibHCellDeriveCells(c, n)        ((c)->DeriveCells = n)
#define SetLibHCellHook(c, n)              ((c)->Hook = n)

/* -----
/* LIB_DERIVE_CELL
/* -----
typedef struct LIB_DERIVE_CELL_STRUCT
{
    LIB_CELL MotherCell;
    BLIST     Inputs;
    BDD       Bdd;
}
LIB_DERIVE_CELL_REC, *LIB_DERIVE_CELL;

#define LibDeriveCellMotherCell(c)   (((c)->MotherCell))
#define LibDeriveCellInputs(c)      (((c)->Inputs))
#define LibDeriveCellBdd(c)         (((c)->Bdd))

#define SetLibDeriveCellMotherCell(c,m)  (((c)->MotherCell = m))
#define SetLibDeriveCellInputs(c,m)    (((c)->Inputs = m))
#define SetLibDeriveCellBdd(c,m)      (((c)->Bdd = m))

/* -----
/* Hooked structure related to the LIB group
/* -----
typedef struct GNL_LIB_STRUCT
{
    char          *Name;           /* name of the library.          */
    char          *SourceFileName; /* File where the library comes from */
    float         Techno;
    int          MaxFanIn;        /* of the combinational LIBC_CELL cells */
    double        AreaBasicEqGate; /* Area of the basic AND2        */
    BLIST         Inverters;       /* List of the LIBC_CELL inverters */
    BLIST         CellsDffs;      /* List of the DFF LIBC_CELL      */
    BLIST         CellsLatches;   /* List of the LATCH LIBC_CELL   */
    /*
    BLIST         Cells3States;   /* List of the 3-States LIBC_CELL */
    BLIST         CellsBuffers;   /* List of the Buffers LIBC_CELL */
    /*
    /* To stores Bdds(BDD) and boolean trees (GNL_NODE) of the cell
    /* function.
    BDD_WS      BddWorkSpace;
    GNL         Gnl;             /* fake gnl to store functions   */
    /* (GNL_NODE) of each LIB_CELL   */
    int          *Hook ;
}
GNL_LIB_REC, *GNL_LIB;

#define GnlHLibName(g)           ((g)->Name)
#define GnlHLibSourceFileName(g) ((g)->SourceFileName)
#define GnlHLibTechno(g)         ((g)->Techno)
#define GnlHLibMaxFanIn(g)       ((g)->MaxFanIn)
#define GnlHLibAreaBasicEqGate(g) ((g)->AreaBasicEqGate)
#define GnlHLibInverters(g)      ((g)->Inverters)
#define GnlHLibCellsDffs(g)     ((g)->CellsDffs)

```

gnllibc.h

```
#define GnlHLibCellsLatches(g)          ((g)->CellsLatches)
#define GnlHLibCells3States(g)           ((g)->Cells3States)
#define GnlHLibCellsBuffers(g)          ((g)->CellsBuffers)

#define GnlHLibBddWorkSpace(g)          ((g)->BddWorkSpace)
#define GnlHLibGnl(g)                  ((g)->Gnl)
#define GnlHLibHook(g)                 ((g)->Hook)

#define SetGnlHLibName(g, l)            ((g)->Name = l)
#define SetGnlHLibSourceFileName(g, l)   ((g)->SourceFileName = l)
#define SetGnlHLibTechno(g, l)          ((g)->Techno = l)
#define SetGnlHLibMaxFanIn(g, l)        ((g)->MaxFanIn = l)
#define SetGnlHLibAreaBasicEqGate(g, l) ((g)->AreaBasicEqGate = l)
#define SetGnlHLibInverters(g, l)       ((g)->Inverters = l)
#define SetGnlHLibCellsDffs(g, l)      ((g)->CellsDffs = l)
#define SetGnlHLibCellsLatches(g, l)    ((g)->CellsLatches = l)
#define SetGnlHLibCells3States(g, l)    ((g)->Cells3States = l)
#define SetGnlHLibCellsBuffers(g, l)    ((g)->CellsBuffers = l)

#define SetGnlHLibBddWorkSpace(g, l)   ((g)->BddWorkSpace = l)
#define SetGnlHLibGnl(g, l)             ((g)->Gnl = l)
#define SetGnlHLibHook(g,h)            ((g)->Hook = h)

/* ----- EOF ----- */

#endif
```

gnlmap.c

```
/*-----*/  
/*-----*/  
/* File: gnlmap.c */  
/* Modifications: - */  
/* Documentation: - */  
/*-----*/  
/* Description: */  
/*-----*/  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
#include "gnloption.h"  
  
#include "blist.e"  
#include "libutil.e"  
  
/*-----*/  
/* For statistics. */  
/*-----*/  
#undef STAT  
int G_NB_TARGET_FUNC = 0;  
int G_ROBDD_BUILT = 0;  
int G_NB_CELL_FOUND = 0;  
float G_WireCapa;  
  
/*-----*/  
/* DEFINE */  
/*-----*/  
#define MAX_AREA_VALUE 1000000000.0  
#define MAX_POWER_VALUE 1000000000.0  
#define MAX_NL_DELAY_VALUE 1000000000.0  
#define MAX_DEPTH_VALUE 1000000000  
#define MAX_NET_VALUE 1000000000  
#define GNL_DEFAULT_FANIN_CELL 6  
  
#define MAX_FACTORIZE_FUNC 100000  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern GNL_ENV G_GnlEnv;  
extern BDD_PTR GetBddPtrFromBdd ();  
extern int CmpVarSignature ();
```

gnlmap.c

```
extern double      GnlGetGnlMapArea () ;

/*-----*/
/* STATIC GLOBAL VARIABLES */
/*-----*/
static GNL  G_Gnl;
static GNL  G_LibGnl;
static GNL_LIB    G_GnlLib;
static BLIST     G_LibInputs;
static LIB_CELL   G_BestInverter;
static int   G_Effort;
static int   G_PrintModuleCounter = 0;

static int   G_MaxCellSizeSupport;

/*-----*/
LIBC_LIB     G_GnlLibc;

/*-----*/
/* FORWARD */
/*-----*/
GNL_STATUS  GnlBestAreaMapNode ();
GNL_STATUS  GnlBestDepthMapNode ();
GNL_STATUS  GnlBestPowerMapNode ();
GNL_STATUS  GnlBestNLDelayMapNode ();
GNL_STATUS  GnlBestNetMapNode ();
GNL_VAR      GnlGetOriginalVar ();

/*-----*/
/* GnlMapNodeInfoCreate */
/*-----*/
/* Create structure in order to store informations for the mapping phase*/
/*-----*/
GNL_STATUS GnlMapNodeInfoCreate (MapNodeInfo)
    GNL_MAP_NODE_INFO    *MapNodeInfo;
{

    if ((*MapNodeInfo = (GNL_MAP_NODE_INFO)
        calloc (1, sizeof (GNL_MAP_NODE_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlCreateSNode */
/*-----*/
GNL_STATUS GnlCreateSNode (SNode)
    GNL_SNODE      *SNode;
{

    if ((*SNode = (GNL_SNODE)calloc (1, sizeof(GNL_SNODE_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

gnlmap.c

```
/*-----*/
/* GnlInsertWireNodeInNode */ */
/*-----*/
GNL_STATUS GnlInsertWireNodeInNode (Gnl, Node, NewNode)
    GNL           Gnl;
    GNL_NODE Node;
    GNL_NODE *NewNode;
{
    int          i;
    GNL_NODE NodeI;
    GNL_NODE NewNodeI;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_WIRE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        if (GnlCreateNode (Gnl, GNL_WIRE, NewNode))
            return (GNL_MEMORY_FULL);

        /* The son of a WIRE node is of type GNL_NODE. */
        SetGnlNodeSons (*NewNode, (BLIST)Node);
        SetGnlNodeLineNumber (*NewNode, GnlNodeLineNumber (Node));
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlInsertWireNodeInNode (Gnl, NodeI, &NewNodeI))
            return (GNL_MEMORY_FULL);
        BListElt (GnlNodeSons (Node), i) = (int)NewNodeI;
    }

    if (GnlCreateNode (Gnl, GNL_WIRE, NewNode))
        return (GNL_MEMORY_FULL);

    /* The son of a WIRE node is of type GNL_NODE. */
    SetGnlNodeSons (*NewNode, (BLIST)Node);
    SetGnlNodeLineNumber (*NewNode, GnlNodeLineNumber (Node));

    return (GNL_OK);
}

/*-----*/
/* GetBestAreaLibInverter */ */
/*-----*/
/* This function looks among the inverters provided in the library */
/* 'GnlLib' and returns the one with the minimum area. */
/*-----*/
LIB_CELL GetBestAreaLibInverter (GnlLib)
    GNL_LIB      GnlLib;
{
```

gnlmap.c

```
int i;
LIB_DERIVE_CELL CellI;
LIB_CELL MotherCellI;
float Area;
LIB_CELL BestCell;

Area = MAX_AREA_VALUE;
for (i=0; i<BListSize (GnlHLibInverters (GnlLib)); i++)
{
    CellI = (LIB_DERIVE_CELL)BListElt (GnlHLibInverters (GnlLib), i);
    MotherCellI = LibDeriveCellMotherCell (CellI);
    if (LibHCellArea (MotherCellI) < Area)
    {
        Area = LibHCellArea (MotherCellI);
        BestCell = MotherCellI;
    }
}
return (BestCell);
}

/*-----
/* GetBestPowerLibInverter
/*-----
/* This function looks among the inverters provided in the library      */
/* 'GnlLib' and returns the one with the minimum Internal Power and with*/
/* minimum area.                                                       */
/*-----*/
LIB_CELL GetBestPowerLibInverter (GnlLib)
    GNL_LIB GnlLib;
{
    int i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL MotherCellI;
    float Power;
    float Area;
    LIB_CELL BestCell;

    Power = MAX_POWER_VALUE;
    Area = MAX_AREA_VALUE;
    for (i=0; i<BListSize (GnlHLibInverters (GnlLib)); i++)
    {
        CellI = (LIB_DERIVE_CELL)BListElt (GnlHLibInverters (GnlLib), i);
        MotherCellI = LibDeriveCellMotherCell (CellI);
        if (LibHCellInternalPower (MotherCellI) < Power)
        {
            Power = LibHCellInternalPower (MotherCellI);
            Area = LibHCellArea (MotherCellI);
            BestCell = MotherCellI;
            continue;
        }

        if ((LibHCellInternalPower (MotherCellI) == Power) &&
            (LibHCellArea (MotherCellI) < Area))
        {

```

gnlmap.c

```
Power = LibHCellInternalPower (MotherCellI);
Area = LibHCellArea (MotherCellI);
BestCell = MotherCellI;
continue;
}
}

return (BestCell);
}

/*-----
/* GnlInsertWireNodes
*/
/*-----
/* This function creates GNL_NODE objects (GNL_WIRE) for each net of the*/
/* Gnl. This can be useful for th technology mapping in order to store */
/* the same information on net than on gate. */
*/
GNL_STATUS GnlInsertWireNodes (Gnl)
    GNL        Gnl;
{
    int          i;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE   NewNode;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        if (GnlInsertWireNodeInNode (Gnl, GnlFunctionOnSet (FunctionI),
                                     &NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlFunctionOnSet (FunctionI, NewNode);
    }

    return (GNL_OK);
}

/*-----
/* GnlAndifyNodePropagate
*/
/*-----
/* This function transform a GNL_NODE 'Node' into a new physical tree */
/* where all operators will be either GNL_AND or GNL_NOT. Moreover */
/* inverters propagation is performed thru the argument 'DoNegation' in */
/* to simplify the whole logic. */
*/
GNL_STATUS GnlAndifyNodePropagate (Gnl, Node, DoNegation, NewNode)
    GNL        Gnl;
    GNL_NODE   Node;
    int         DoNegation;
    GNL_NODE   *NewNode;
{
    int          i;
    BLIST       SonList;
    GNL_NODE   SonI;
```

gnlmap.c

```
BLIST      Sons;
GNL_NODE   SNode;
GNL_NODE   VarNode;
GNL_NODE   AndNode;

if (Node == NULL)
{
    *NewNode = NULL;
    return (GNL_OK);
}

if (GnlCreateNode (Gnl, NULL, NewNode))
    return (GNL_MEMORY_FULL);

switch (GnlNodeOp (Node)) {
    case GNL_AND:
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);
        if (DoNegation)
        {
            /* GNL_NOT of GNL_AND ... */ /* */
            SetGnlNodeOp (*NewNode, GNL_NOT);
            SetGnlNodeSons (*NewNode, SonList);
            if (GnlCreateNode (Gnl, GNL_AND, &AndNode))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)AndNode))
                return (GNL_MEMORY_FULL);

            if (BListCreateWithSize (1, &SonList))
                return (GNL_MEMORY_FULL);
            SetGnlNodeSons (AndNode, SonList);

            Sons = GnlNodeSons (Node);
            for (i=0; i<BListSize (Sons); i++)
            {
                SonI = (GNL_NODE)BListElt (Sons, i);
                if (GnlAndifyNodePropagate (Gnl, SonI,
                                            0, &SNode))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (SonList, (int)SNode))
                    return (GNL_MEMORY_FULL);
            }
            return (GNL_OK);
        }
        else /* We create the inverter with a Nand gate */ /* */
        {
            SetGnlNodeOp (*NewNode, GNL_AND);
            SetGnlNodeSons (*NewNode, SonList);
            Sons = GnlNodeSons (Node);
            for (i=0; i<BListSize (Sons); i++)
            {
                SonI = (GNL_NODE)BListElt (Sons, i);
                if (GnlAndifyNodePropagate (Gnl, SonI,
                                            0, &SNode))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (SonList, (int)SNode))
                    return (GNL_MEMORY_FULL);
            }
        }
    }
}
```

```

                return (GNL_MEMORY_FULL);
            }
            return (GNL_OK);
        }
    break;

case GNL_OR:
    if (BListCreateWithSize (1, &SonList))
        return (GNL_MEMORY_FULL);
    if (!DoNegation) /* We create an inverter */
    {
        SetGnlNodeOp (*NewNode, GNL_NOT);
        if (GnlCreateNode (Gnl, GNL_AND, &AndNode))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (SonList, (int)AndNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, SonList);
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);

        Sons = GnlNodeSons (Node);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlAndifyNodePropagate (Gnl, SonI,
                                         1, &SNode))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)SNode))
                return (GNL_MEMORY_FULL);
        }
        SetGnlNodeSons (AndNode, SonList);
        return (GNL_OK);
    }
else
{
    SetGnlNodeOp (*NewNode, GNL_AND);
    Sons = GnlNodeSons (Node);
    for (i=0; i<BListSize (Sons); i++)
    {
        SonI = (GNL_NODE)BListElt (Sons, i);
        if (GnlAndifyNodePropagate (Gnl, SonI,
                                    1, &SNode))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (SonList, (int)SNode))
            return (GNL_MEMORY_FULL);
    }
    SetGnlNodeSons (*NewNode, SonList);
    return (GNL_OK);
}
break;

case GNL_NOT:
    Sons = GnlNodeSons (Node);
    SonI = (GNL_NODE)BListElt (Sons, 0);
    if (DoNegation)
        return (GnlAndifyNodePropagate (Gnl, SonI,
                                       0, NewNode));

```

gnlmap.c

```
else
    return (GnlAndifyNodePropagate (Gnl, SonI,
                                    1, NewNode));
break;

case GNL_VARIABLE:
    if (!DoNegation)
    {
        SetGnlNodeOp (*NewNode, GNL_VARIABLE);
        SetGnlNodeSons (*NewNode, GnlNodeSons (Node));
        return (GNL_OK);
    }
    else
    {
        SetGnlNodeOp (*NewNode, GNL_NOT);
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);

        if (GnlCreateNode (Gnl, GNL_VARIABLE, &VarNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (VarNode, GnlNodeSons (Node));
        if (BListAddElt (SonList, (int)VarNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, SonList);
        return (GNL_OK);
    }
break;

case GNL_CONSTANTE:
    SetGnlNodeOp (*NewNode, GNL_CONSTANTE);
    if (DoNegation)
        SetGnlNodeSons (*NewNode, (BLIST) !GnlNodeSons (Node));
    else
        SetGnlNodeSons (*NewNode, (BLIST) GnlNodeSons (Node));
    return (GNL_OK);

default:
    fprintf (stderr, "4:Unknown node operator\n");
    exit (1);
}

/*
 * GnlAndifyNode
 */
/* This function transform a GNL_NODE 'Node' into a new physical tree */
/* where all operators will be either GNL_AND or GNL_NOT. In this case */
/* GNL_OR are transformed by De Morgan law and no inverter propagation */
/* is performed. This is use for the best area option because we can use*/
/* input polarities with this. */
*/
GNL_STATUS GnlAndifyNode (Gnl, Node, NewNode)
    GNL          Gnl;
    GNL_NODE     Node;
    GNL_NODE     *NewNode;
```

```

{
    int          i;
    BLIST        SonList;
    GNL_NODE    SonI;
    BLIST        Sons;
    GNL_NODE    SNode;
    GNL_NODE    SNodeNot;
    GNL_NODE    VarNode;
    GNL_NODE    AndNode;

    if (Node == NULL)
    {
        *NewNode = NULL;
        return (GNL_OK);
    }

    if (GnlCreateNode (Gnl, NULL, NewNode))
        return (GNL_MEMORY_FULL);

    switch (GnlNodeOp (Node)) {
        case GNL_AND:
            if (BListCreateWithSize (1, &SonList))
                return (GNL_MEMORY_FULL);
            SetGnlNodeOp (*NewNode, GNL_AND);
            SetGnlNodeSons (*NewNode, SonList);
            Sons = GnlNodeSons (Node);
            for (i=0; i<BListSize (Sons); i++)
            {
                SonI = (GNL_NODE)BListElt (Sons, i);
                if (GnlAndifyNode (Gnl, SonI, &SNode))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (SonList, (int)SNode))
                    return (GNL_MEMORY_FULL);
            }
            return (GNL_OK);

        case GNL_OR:
            if (BListCreateWithSize (1, &SonList))
                return (GNL_MEMORY_FULL);
            SetGnlNodeOp (*NewNode, GNL_NOT);
            if (GnlCreateNode (Gnl, GNL_AND, &AndNode))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)AndNode))
                return (GNL_MEMORY_FULL);
            SetGnlNodeSons (*NewNode, SonList);
            if (BListCreateWithSize (1, &SonList))
                return (GNL_MEMORY_FULL);

            Sons = GnlNodeSons (Node);
            for (i=0; i<BListSize (Sons); i++)
            {
                SonI = (GNL_NODE)BListElt (Sons, i);
                if (GnlAndifyNode (Gnl, SonI, &SNode))
                    return (GNL_MEMORY_FULL);
                if (GnlCreateNodeNot (Gnl, SNode, &SNodeNot))

```

```

        return (GNL_MEMORY_FULL);
    if (BListAddElt (SonList, (int)SNodeNot))
        return (GNL_MEMORY_FULL);
    }
SetGnlNodeSons (AndNode, SonList);
return (GNL_OK);

case GNL_NOT:
    if (BListCreateWithSize (1, &SonList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeOp (*NewNode, GNL_NOT);
    Sons = GnlNodeSons (Node);
    SonI = (GNL_NODE)BListElt (Sons, 0);
    if (GnlAndifyNode (Gnl, SonI, &SNode))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (SonList, (int)SNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*NewNode, SonList);
    return (GNL_OK);

case GNL_VARIABLE:
    SetGnlNodeOp (*NewNode, GNL_VARIABLE);
    SetGnlNodeSons (*NewNode, GnlNodeSons (Node));
    return (GNL_OK);

case GNL_CONSTANTE:
    SetGnlNodeOp (*NewNode, GNL_CONSTANTE);
    SetGnlNodeSons (*NewNode, (BLIST)GnlNodeSons (Node));
    return (GNL_OK);

default:
    fprintf (stderr, "1:Unknown node operator\n");
    exit (1);
}

/*
 * GnlAndifyPropagate
 */
/*
 * This function transforms the current Gnl netlist into a netlist using*
 * only nodes of type GNL_NOT, GNL_AND, GNL_VARIABLE or GNL_CONSTANTE   */
/* Moreover, Inverters propagations are performed in order to simplify   */
/* the Gnl description.                                              */
/*
GNL_STATUS GnlAndifyPropagate (Gnl)
    GNL      Gnl;
{
    int          i;
    char         *CopyName;
    GNL_VAR     Vari;
    GNL_FUNCTION FunctionI;
    GNL_NODE    NewOnSet;
    GNL_NODE    NewDCSet;

```

gnlmap.c

```
BLIST      NewListFunctions;
int        j;
BLIST      NodesSegments;
GNL       NewGnl;

if ((NewGnl = (GNL)malloc (1, sizeof(GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

SetGnlNbIn (NewGnl, GnlNbIn (Gnl));
SetGnlNbOut (NewGnl, GnlNbOut (Gnl));
SetGnlNbLocal (NewGnl, GnlNbLocal (Gnl));

SetGnlInputs (NewGnl, GnlInputs (Gnl));
SetGnlOutputs (NewGnl, GnlOutputs (Gnl));
SetGnlLocals (NewGnl, GnlLocals (Gnl));

SetGnlHashNames (NewGnl, GnlHashNames (Gnl));

if (BListCreate (&NodesSegments))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments (NewGnl, NodesSegments);
SetGnlFirstNode (NewGnl, NULL);
SetGnlLastNode (NewGnl, NULL);

SetGnlComponents (NewGnl, GnlComponents (Gnl));

if (BListCreate (&NewListFunctions))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " Structuring (1) for mapping...");
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (GnlAndifyNodePropagate (NewGnl, GnlFunctionOnSet (FunctionI),
                                0, &NewOnSet))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (NewListFunctions, (int)NewOnSet))
        return (GNL_MEMORY_FULL);
}

fprintf (stderr, "\n");

/* Deleting all the nodes from the gnl 'Gnl'. */  

GnlFreeNodesSegments (Gnl);

i = 0;
for (j=0; j<BListSize (NewListFunctions); j++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    NewOnSet = (GNL_NODE)BListElt (NewListFunctions, j);
    SetGnlFunctionOnSet (FunctionI, NewOnSet);
```

gnlmap.c

```
i++;
}
BListQuickDelete (&NewListFunctions);

SetGnlNodesSegments (Gnl, GnlNodesSegments (NewGnl));
SetGnlFirstNode (Gnl, GnlFirstNode (NewGnl));
SetGnlLastNode (Gnl, GnlLastNode (NewGnl));

free ((char*)NewGnl);

return (GNL_OK);
}

/*-----
/* GnlAndify
*/
/* This function transforms the current Gnl netlist into a netlist using*/
/* only nodes of type GNL_NOT, GNL_AND, GNL_VARIABLE or GNL_CONSTANTE. */
/* No Inverters propagation are performed so the description is always */
/* bigger than the original one. */
*/
GNL_STATUS GnlAndify (Gnl)
    GNL      Gnl;
{
    int          i;
    char         *CopyName;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE   NewOnSet;
    GNL_NODE   NewDCSet;
    BLIST      NewListFunctions;
    int          j;
    BLIST      NodesSegments;
    GNL      NewGnl;

    if ((NewGnl = (GNL)malloc (1, sizeof(GNL_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlNbIn (NewGnl, GnlNbIn (Gnl));
    SetGnlNbOut (NewGnl, GnlNbOut (Gnl));
    SetGnlNbLocal (NewGnl, GnlNbLocal (Gnl));

    SetGnlInputs (NewGnl, GnlInputs (Gnl));
    SetGnlOutputs (NewGnl, GnlOutputs (Gnl));
    SetGnlLocals (NewGnl, GnlLocals (Gnl));

    SetGnlHashNames (NewGnl, GnlHashNames (Gnl));

    if (BListCreate (&NodesSegments))
        return (GNL_MEMORY_FULL);
    SetGnlNodesSegments (NewGnl, NodesSegments);
    SetGnlFirstNode (NewGnl, NULL);
    SetGnlLastNode (NewGnl, NULL);
```

gnlmap.c

```
SetGnlComponents (NewGnl, GnlComponents (Gnl));

if (BListCreate (&NewListFunctions))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (Vari);

    if (GnlAndifyNode (NewGnl, GnlFunctionOnSet (FunctionI),
                       &NewOnSet))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (NewListFunctions, (int)NewOnSet))
        return (GNL_MEMORY_FULL);
}

GnlFreeNodesSegments (Gnl);

i = 0;
for (j=0; j<BListSize (NewListFunctions); j++)
{
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (Vari);
    NewOnSet = (GNL_NODE)BListElt (NewListFunctions, j);
    SetGnlFunctionOnSet (FunctionI, NewOnSet);
    i++;
}
BListQuickDelete (&NewListFunctions);

SetGnlNodesSegments (Gnl, GnlNodesSegments (NewGnl));
SetGnlFirstNode (Gnl, GnlFirstNode (NewGnl));
SetGnlLastNode (Gnl, GnlLastNode (NewGnl));

free ((char*)NewGnl);

return (GNL_OK);
}

/*
 * GnlNAndifyNode
 */
/*
 * This function transform a GNL_NODE 'Node' into a new physical tree
 * where all operators will be either GNL_NAND or GNL_NOT. In this case
 * GNL_OR are transformed by De Morgan law and no inverter propagation
 * is performed. This is use for the best area option because we can use
 * input polarities with this.
 */
GNL_STATUS GnlNAndifyNode (Gnl, Node, NewNode)
    GNL          Gnl;
    GNL_NODE    Node;
    GNL_NODE    *NewNode;
{
    int          i;
    BLIST       SonList;
```

gnlmap.c

```
GNL_NODE      SonI;
BLIST         Sons;
GNL_NODE      SNode;
GNL_NODE      SNodeNot;
GNL_NODE      VarNode;
GNL_NODE      AndNode;
GNL_NODE      NAndNode;

if (Node == NULL)
{
    *NewNode = NULL;
    return (GNL_OK);
}

if (GnlCreateNode (Gnl, NULL, NewNode))
    return (GNL_MEMORY_FULL);

switch (GnlNodeOp (Node)) {
    case GNL_AND:
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);
        SetGnlNodeOp (*NewNode, GNL_NAND);
        SetGnlNodeSons (*NewNode, SonList);

        Sons = GnlNodeSons (Node);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlNAndifyNode (Gnl, SonI, &SNode))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)SNode))
                return (GNL_MEMORY_FULL);
        }
        if (GnlCreateNodeNot (Gnl, *NewNode, &NAndNode))
            return (GNL_MEMORY_FULL);
        *NewNode = NAndNode;
        return (GNL_OK);

    case GNL_OR:
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);
        SetGnlNodeOp (*NewNode, GNL_NAND);
        SetGnlNodeSons (*NewNode, SonList);

        Sons = GnlNodeSons (Node);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlNAndifyNode (Gnl, SonI, &SNode))
                return (GNL_MEMORY_FULL);
            if (GnlCreateNodeNot (Gnl, SNode, &SNodeNot))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)SNodeNot))
                return (GNL_MEMORY_FULL);
        }
        return (GNL_OK);
}
```

```

case GNL_NOT:
    if (BListCreateWithSize (1, &SonList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeOp (*NewNode, GNL_NOT);
    Sons = GnlNodeSons (Node);
    SonI = (GNL_NODE)BListElt (Sons, 0);
    if (GnlNAndifyNode (Gnl, SonI, &SNode))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (SonList, (int)SNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*NewNode, SonList);
    return (GNL_OK);

case GNL_VARIABLE:
    SetGnlNodeOp (*NewNode, GNL_VARIABLE);
    SetGnlNodeSons (*NewNode, GnlNodeSons (Node));
    return (GNL_OK);

case GNL_CONSTANTE:
    SetGnlNodeOp (*NewNode, GNL_CONSTANTE);
    SetGnlNodeSons (*NewNode, (BLIST)GnlNodeSons (Node));
    return (GNL_OK);

default:
    fprintf (stderr, "2:Unknown node operator\n");
    exit (1);
}

/*
 * GnlAndify
 */
/*
 * This function transforms the current Gnl netlist into a netlist using*
 * only nodes of type GNL_NOT, GNL_NAND, GNL_VARIABLE or GNL_CONSTANTE. *
 * No Inverters propagation are performed so the description is always   *
 * bigger than the original one.
 */
GNL_STATUS GnlNAndify (Gnl)
{
    GNL Gnl;
    int i;
    char *CopyName;
    GNL_VAR VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE NewOnSet;
    GNL_NODE NewDCSet;
    BLIST NewListFunctions;
    int j;
    BLIST NodesSegments;
    GNL NewGnl;
}

```

```

if ((NewGnl = (GNL)malloc (1, sizeof (GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

SetGnlNbIn (NewGnl, GnlNbIn (Gnl));
SetGnlNbOut (NewGnl, GnlNbOut (Gnl));
SetGnlNbLocal (NewGnl, GnlNbLocal (Gnl));

SetGnlInputs (NewGnl, GnlInputs (Gnl));
SetGnlOutputs (NewGnl, GnlOutputs (Gnl));
SetGnlLocals (NewGnl, GnlLocals (Gnl));

SetGnlHashNames (NewGnl, GnlHashNames (Gnl));

if (BListCreate (&NodesSegments))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments (NewGnl, NodesSegments);
SetGnlFirstNode (NewGnl, NULL);
SetGnlLastNode (NewGnl, NULL);

SetGnlComponents (NewGnl, GnlComponents (Gnl));

if (BListCreate (&NewListFunctions))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (GnlNAndifyNode (NewGnl, GnlFunctionOnSet (FunctionI),
                        &NewOnSet))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (NewListFunctions, (int)NewOnSet))
        return (GNL_MEMORY_FULL);
}

GnlFreeNodesSegments (Gnl);

i = 0;
for (j=0; j<BListSize (NewListFunctions); j++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    NewOnSet = (GNL_NODE)BListElt (NewListFunctions, j);
    SetGnlFunctionOnSet (FunctionI, NewOnSet);
    i++;
}
BListQuickDelete (&NewListFunctions);

SetGnlNodesSegments (Gnl, GnlNodesSegments (NewGnl));
SetGnlFirstNode (Gnl, GnlFirstNode (NewGnl));
SetGnlLastNode (Gnl, GnlLastNode (NewGnl));

free ((char*)NewGnl);

```

gnlmap.c

```
    return (GNL_OK);
}

/*-----
/* GnlBalanceComp                               */
/*-----*/
/* Comparison function to check balancing between two Trees.      */
/*-----*/
static int GnlBalanceComp (Node1, Node2)
    GNL_NODE *Node1;
    GNL_NODE *Node2;
{
    int          Critic1;
    int          Critic2;

    Critic1 = GnlNodeMaxDepth (*Node1);
    Critic2 = GnlNodeMaxDepth (*Node2);

    if (Critic1 < Critic2)
        return (-1);

    if (Critic1 == Critic2)
        return (0);

    return (1);
}

/*-----
/* GnlCreateBalancedNode                         */
/*-----*/
/* This function modifies the GNL_NODE 'Node' structure in order to get */
/* a balanced binary tree. Balancing is done according to the cost      */
/* function 'GnlBalanceComp'.                                         */
/*-----*/
GNL_STATUS GnlCreateBalancedNode (Gnl, Criter, Node)
    GNL           Gnl;
    int           Criter;
    GNL_NODE     Node;
{
    int          i;
    BLIST       Sons;
    GNL_NODE   Son1;
    GNL_NODE   Son2;
    GNL_NODE  NewNode;
    BLIST       SonList;

    if ((GnlNodeOp (Node) != GNL_AND) &&
        (GnlNodeOp (Node) != GNL_NOT) &&
        (GnlNodeOp (Node) != GNL_OR))
        return (GNL_OK);

    Sons = GnlNodeSons (Node);

    /* Nothing to do if the number of sons is already two             */
}
```

gnlmap.c

```
if (BListSize (Sons) == 2)
    return (GNL_OK);

while (BListSize (Sons) > 2)
{
    /* 'Criter' = 0 --> Depth Minimization */ 
    if (!Criter)
        qsort (BListAdress (Sons), BListSize(Sons), sizeof (GNL_NODE),
               GnlBalanceComp);

    Son1 = (GNL_NODE)BListElt (Sons, 0);
    Son2 = (GNL_NODE)BListElt (Sons, 1);
    BListDelShift (Sons, 1);
    BListDelShift (Sons, 1);

    /* we build the Function Node of the two least critical sons */
    if (GnlCreateNode (Gnl, GnlNodeOp (Node), &NewNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &SonList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (SonList, (int)Son1))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (SonList, (int)Son2))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (NewNode, SonList);

    if (BListAddElt (Sons, (int)NewNode))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlMakeBalanceBinaryNode
/*-----*/
/* This function creates a new node 'NewNode' from 'Node' which is a */
/* balanced binary tree. */
/*-----*/
GNL_STATUS GnlMakeBalanceBinaryNode (Gnl, Node, Criter, NewNode)
    GNL          Gnl;
    GNL_NODE     Node;
    int          Criter;           /* 'Criter' = 1 -> Area, 0 -> depth */
    GNL_NODE     *NewNode;
{
    int          i;
    BLIST        SonList;
    GNL_NODE     SonI;
    BLIST        Sons;
    GNL_NODE     SNode;
    GNL_NODE     VarNode;

    if (Node == NULL)
    {
        *NewNode = NULL;
```

```

        return (GNL_OK);
    }

switch (GnlNodeOp (Node)) {
    case GNL_AND:
    case GNL_OR:
        if (GnlCreateNode (Gnl, GnlNodeOp (Node), NewNode))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);
        Sons = GnlNodeSons (Node);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlMakeBalanceBinaryNode (Gnl, SonI, Criter, &SNode))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (SonList, (int)SNode))
                return (GNL_MEMORY_FULL);
        }
        SetGnlNodeSons (*NewNode, SonList);
        if (GnlCreateBalancedNode (Gnl, Criter, *NewNode))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
        break;

    case GNL_NOT:
        if (GnlCreateNode (Gnl, GNL_NOT, NewNode))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (1, &SonList))
            return (GNL_MEMORY_FULL);
        Sons = GnlNodeSons (Node);
        SonI = (GNL_NODE)BListElt (Sons, 0);
        if (GnlMakeBalanceBinaryNode (Gnl, SonI, Criter, &SNode))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (SonList, (int)SNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, SonList);
        return (GNL_OK);
        break;

    case GNL_VARIABLE:
        if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, GnlNodeSons (Node));
        return (GNL_OK);
        break;

    case GNL_CONSTANTE:
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, (BLIST)GnlNodeSons (Node));
        return (GNL_OK);
        break;
}

```

```

default:
    fprintf (stderr, "3:Unknown node operator\n");
    exit (1);
}

/*
 * GnlMakeBalanceBinary
 */
/*
 * This modifies the GNL 'Gnl' and replaces all its functions by *
 * balanced binary trees.                                         */
GNL_STATUS GnlMakeBalanceBinary (Gnl, Criter)
    GNL      Gnl;
    int      Criter;
{
    int      i;
    char     *CopyName;
    GNL_VAR  VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE NewOnSet;
    GNL_NODE NewDCSet;
    int      j;
    BLIST    NodesSegments;
    GNL      NewGnl;
    BLIST    ListNodesSegments;
    GNL_NODE SegmentI;

    if ((NewGnl = (GNL)malloc (1, sizeof(GNL_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlNbIn (NewGnl, GnlNbIn (Gnl));
    SetGnlNbOut (NewGnl, GnlNbOut (Gnl));
    SetGnlNbLocal (NewGnl, GnlNbLocal (Gnl));

    SetGnlInputs (NewGnl, GnlInputs (Gnl));
    SetGnlOutputs (NewGnl, GnlOutputs (Gnl));
    SetGnlLocals (NewGnl, GnlLocals (Gnl));

    SetGnlHashNames (NewGnl, GnlHashNames (Gnl));

    if (BListCreate (&NodesSegments))
        return (GNL_MEMORY_FULL);
    SetGnlNodesSegments (NewGnl, NodesSegments);
    SetGnlFirstNode (NewGnl, NULL);
    SetGnlLastNode (NewGnl, NULL);

    SetGnlComponents (NewGnl, GnlComponents (Gnl));

    fprintf (stderr, " Structuring (2) for mapping...");
```

gnlmap.c

```
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (GnlMakeBalanceBinaryNode (NewGnl, GnlFunctionOnSet (FunctionI),
                                  Criter, &NewOnSet))
        return (GNL_MEMORY_FULL);

    GnlFreeNodeSons (GnlFunctionOnSet (FunctionI));
    GnlFreeNodeSons (GnlFunctionDCSet (FunctionI));

    SetGnlFunctionOnSet (FunctionI, NewOnSet);
}
fprintf (stderr, "\n");

ListNodesSegments = GnlNodesSegments (Gnl);

for (i=0; i<BListSize (ListNodesSegments); i++)
{
    SegmentI = (GNL_NODE)BListElt (ListNodesSegments, i);
    free ((char*)SegmentI);
}
SetGnlFirstNode (Gnl, NULL);
SetGnlLastNode (Gnl, NULL);

/*
i = 0;
for (j=0; j<BListSize (NewListFunctions); j++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    NewOnSet = (GNL_NODE)BListElt (NewListFunctions, j);
    SetGnlFunctionOnSet (FunctionI, NewOnSet);
    i++;
}
BListQuickDelete (&NewListFunctions);
*/
SetGnlNodesSegments (Gnl, GnlNodesSegments (NewGnl));

free ((char*)NewGnl);

return (GNL_OK);
}

/*-----
/* GnlResetGnlNodeHook
/*-----
/* We recursively reset to NULL the field Hook of node 'Node'.
/*-----
static void GnlResetGnlNodeHook (Node)
    GNL_NODE Node;
{
    int           i;
```

gnlmap.c

```
GNL_NODE SonI;

if (GnlNodeOp (Node) == GNL_CONSTANTE)
{
    SetGnlNodeHook (Node, NULL);
    return ;
}

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    SetGnlNodeHook (Node, NULL);
    return ;
}

SetGnlNodeHook (Node, NULL);

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
    GnlResetGnlNodeHook (SonI);
}

/*
 *-----*
 * GnlAddMapNodeInfo
 *-----*/
/* This function adds recursively a data structure 'GNL_MAP_NODE_INFO' */
/* to GNL_NODE 'Node' and hooks it on the field 'GnlNodeHook (Node)'.
 *-----*/
GNL_STATUS GnlAddMapNodeInfo (Node)
    GNL_NODE Node;
{
    int                 i;
    GNL_MAP_NODE_INFO  NewMapNodeInfo;
    GNL_NODE           SonI;
    GNL_SNODE          SiNode;
    GNL_SNODE          PSNode;

    /* If we already pass thru this node */
    if ((GnlNodeHook (Node) != NULL) &&
        (GnlMapNodeInfoSNodes (Node) != NULL))
        return (GNL_OK);

    if (GnlNodeHook (Node) == NULL)
    {
        if (GnlMapNodeInfoCreate (&NewMapNodeInfo))
            return (GNL_MEMORY_FULL);
        SetGnlNodeHook (Node, NewMapNodeInfo);
    }

    if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
        (GnlNodeOp (Node) == GNL_CONSTANTE))
    {
        return (GNL_OK);
```

gnlmap.c

```
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

    if (GnlAddMapNodeInfo (SonI))
        return (GNL_MEMORY_FULL);

    if (GnlCreateSNode (&SiNode))
        return (GNL_MEMORY_FULL);
    SetGnlSNodeNode (SiNode, SonI);

    if (GnlMapNodeInfoSNodes (Node) == NULL)
    {
        SetGnlMapNodeInfoSNodes (Node, SiNode);
    }
    else
        SetGnlSNodeNext (PSNode, SiNode);

    PSNode = SiNode;
}

return (GNL_OK);
}

/*-----
/* GnlFreeMapSNodes
*/
void GnlFreeMapSNodes (SNode)
    GNL_SNODE      SNode;

{
    if (SNode == NULL)
        return;

    GnlFreeMapSNodes (GnlSNodeNext (SNode));
    free (SNode);
}

/*-----
/* GnlFreeSNodeOfMapNodeInfo
*/
void GnlFreeSNodeOfMapNodeInfo (Node)
    GNL_NODE Node;
{
    int                  i;
    GNL_NODE            SonI;

    /* If we already pass thru this node */
    if (GnlMapNodeInfoSNodes (Node) == NULL)
        return;

    if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
        (GnlNodeOp (Node) == GNL_CONSTANTE))
```

gnlmap.c

```
{  
    return ;  
}  
  
GnlFreeMapSNodes (GnlMapNodeInfoSNodes (Node)) ;  
SetGnlMapNodeInfoSNodes (Node, NULL) ;  
  
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
{  
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);  
    GnlFreeSNodeOfMapNodeInfo (SonI);  
}  
  
}  
  
/*-----*/  
/* GnlBuildDirectBddOnMapNode */  
/*-----*/  
/* This procedure builds a direct Bdd on the node 'Node' and stores it */  
/* in the field 'GnlMapNodeInfoBdd (Node)'. This procedure takes as */  
/* Bdd of a Var the Bdd of its associated lib. var which must exist. */  
/*-----*/  
BDD_STATUS GnlBuildDirectBddOnMapNode (Node)  
{  
    GNL_NODE      Node;  
  
    int           i;  
    BDD_STATUS    Status;  
    BLIST         Sons;  
    GNL_NODE     NodeI;  
    BDD          NewBdd;  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        GNL_VAR      Var;  
  
        /* We access the associated library var of this node. */  
        Var = GnlMapNodeInfoLibVar (Node);  
  
        /* The Bdd for this node is the Bdd of the library var */  
        SetGnlMapNodeInfoBdd (Node, UseBdd (GnlVarBdd (Var)));  
        return (BDD_OK);  
    }  
  
    if (GnlTag (G_Gnl) == GnlNodeTag (Node))  
    {  
        GNL_VAR      Var;  
  
        Var = GnlMapNodeInfoLibVar (Node);  
        SetGnlMapNodeInfoBdd (Node, UseBdd (GnlVarBdd (Var)));  
        return (BDD_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
    {  
        if (GnlNodeSons (Node) == (BLIST)0)  
            SetGnlMapNodeInfoBdd (Node, bdd_zero());  
    }
```

gnlmap.c

```

    else
        SetGnlMapNodeInfoBdd (Node, bdd_one());
    return (BDD_OK);
}

Sons = GnlNodeSons (Node);
for (i=0; i<BListSize (Sons); i++)
{
    NodeI = (GNL_NODE)BListElt (Sons, i);
    if ((Status = GnlBuildDirectBddOnMapNode (NodeI)))
        return (Status);
}

/* All the Bdd of the sons have been computed. Now we compute the Bdd*/
/* of Node. */
if ((Status = GnlBddNodeApply (1, G_LibGnl, Node, &NewBdd)))
    return (Status);

SetGnlMapNodeInfoBdd (Node, NewBdd);

return (BDD_OK);
}

/*
 *-----*
 * GnlGetBestAreaCell
 *-----*
 * This function returns the Best Cell and Best Area from a given list
 * cells. The list of cells is a list of LIB_DERIVE_CELL and all have
 * the same Bdd or !Bdd. A cell having a bdd LibDeriveCellBdd (Cell)
 * which !Bdd needs also an inverter to realize the !. So the area
 * associated to this cell is its own area + area best inverter. If the
 * Bdd LibDeriveCellBdd (Cell) of a cell is Bdd then its area is simply
 * LibHCellArea (LibDeriveCellMotherCell (cell)). */
*-----*
void GnlGetBestAreaCell (ListCells, Bdd, BestCell, BestArea, BestDepth)
    BLIST          ListCells;
    BDD            Bdd;
    LIB_DERIVE_CELL *BestCell;
    float          *BestArea;
    int             *BestDepth;
{
    int              i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL         MotherCellI;
    float           CArea;
    int              CDepth;

    *BestArea = MAX_AREA_VALUE;
    *BestCell = NULL;
    *BestDepth = MAX_DEPTH_VALUE;

    for (i=0; i<BListSize (ListCells); i++)
    {
        /* first pick up the derive cell area thru its mothercell. */
        CellI = (LIB_DERIVE_CELL)BListElt (ListCells, i);
    }
}

```

gnlmap.c

```
MotherCellI = LibDeriveCellMotherCell (CellI);
CArea = LibHCellArea (MotherCellI);
CDepth = 1;

/* If Bdd of the derive cell is the complement of the Bdd of the*/
/* current node then an inverter is mandatory so we increase      */
/* the Area of the cell by the area of the best inverter.      */
/* Corner case: if the cell is itself the best inverter then we */
/* do not do it.                                              */
if ((MotherCellI != G_BestInverter) &&
    (Bdd != LibDeriveCellBdd (CellI)))
{
    CArea += LibHCellArea (G_BestInverter);
    CDepth++;
}

/* storing the new best area.                                     */
if ((CArea < *BestArea) ||
    ((CArea == *BestArea) && (CDepth < *BestDepth)))
{
    *BestArea = CArea;
    *BestCell = CellI;
    *BestDepth = CDepth;
}

}

/*-----
/* GnlCellPowerEstimate                                         */
/*-----*/
float GnlCellPowerEstimate (MotherCell, CellCapa)
    LIB_CELL MotherCell;
    float CellCapa;
{
    float Power;

    Power = LibHCellInternalPower (MotherCell) +
        0.5 * GnlEnvVDD () * GnlEnvVDD () *
        LibHCellChargeProba (MotherCell) * CellCapa;

/*
fprintf (stderr,
    " GATE : %s\n    IPwr = %f    ChargeProba = %f    CellCapa = %f\n",
    LibHCellName (MotherCell), LibHCellInternalPower (MotherCell),
    LibHCellChargeProba (MotherCell), CellCapa);
*/
    return (Power);
}

/*-----
/* GnlGetBestPowerCell                                         */
/*-----*/
```

```

/* This function returns the Best Cell from which we expect the minimum */
/* power consumption. Since we cannot have an accurate estimation at    */
/* level to estimate the correct power comsumption, we try to take the   */
/* cell with the minimum charge probability. This value is given by the */
/* field 'LibHCellChargeProba (MotherCellI)'.                           */
/*-----*/
void GnlGetBestPowerCell (ListCells, Capacitance, Bdd, BestCell,
                         BestPower, BestDepth)
{
    BLIST           ListCells;
    float          Capacitance;
    BDD            Bdd;
    LIB_DERIVE_CELL *BestCell;
    float          *BestPower;
    int             *BestDepth;
{
    int             i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL        MotherCellI;
    float          CPower;
    int             CDepth;
    int             Area;
    int             BestArea;

    *BestPower = MAX_POWER_VALUE;
    *BestCell = NULL;
    *BestDepth = MAX_DEPTH_VALUE;
    BestArea = MAX_AREA_VALUE;

    for (i=0; i<BListSize (ListCells); i++)
    {
        /* first pick up the derive cell area thru its mothercell.      */
        CellI = (LIB_DERIVE_CELL)BListElt (ListCells, i);
        MotherCellI = LibDeriveCellMotherCell (CellI);
        CPower = GnlCellPowerEstimate (MotherCellI, Capacitance);
        CDepth = 1;
        Area = LibHCellArea (MotherCellI);

        /* If Bdd of the derive cell is the complement of the Bdd of the */
        /* current node then an inverter is mandatory so we increase     */
        /* the Area of the cell by the area of the best inverter.       */
        /* Corner case: if the cell is itself the best inverter then we */
        /* do not do it.                                                 */
        if ((MotherCellI != G_BestInverter) &&
            (Bdd != LibDeriveCellBdd (CellI)))
        {
            CPower += GnlCellPowerEstimate (G_BestInverter, Capacitance);
            CDepth++;
            Area += LibHCellArea (G_BestInverter);
        }

        /* storing the new best area.                                     */
        if ((CPower < *BestPower) ||
            ((CPower == *BestPower) && (CDepth < *BestDepth)))
        {
            *BestPower = CPower;
            *BestCell = CellI;
        }
    }
}

```

```

        *BestDepth = CDepth;
        BestArea = Area;
        continue;
    }

    if ((CPower == *BestPower) && (CDepth == *BestDepth) &&
        (Area < BestArea))
    {
        *BestPower = CPower;
        *BestCell = CellI;
        *BestDepth = CDepth;
        BestArea = Area;
        continue;
    }
}

/*
 * GnlGetBestDepthCell
 */
/*
 * This function returns the Best Cell and Best Depth from a given list *
 * cells. The list of cells is a list of LIB_DERIVE_CELL and all have   *
 * the same Bdd or !Bdd. A cell having a bdd LibDeriveCellBdd (Cell)   *
 * which !Bdd needs also an inverter to realize the !. So the Depth   *
 * associated to this cell is its own Depth + inverter Depth. If the   *
 * Bdd LibDeriveCellBdd (Cell) of a cell is Bdd then its depth is simply*
 * 1.                                         */
void GnlGetBestDepthCell (ListCells, Bdd, BestCell, BestArea, BestDepth)
{
    BLIST          ListCells;
    BDD            Bdd;
    LIB_DERIVE_CELL *BestCell;
    float          *BestArea;
    int             *BestDepth;

    int             i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL        MotherCellI;
    float           CArea;
    int             CDepth;

    *BestArea = MAX_AREA_VALUE;
    *BestCell = NULL;
    *BestDepth = MAX_DEPTH_VALUE;

    for (i=0; i<BListSize (ListCells); i++)
    {
        /* first pick up the derive cell area thru its mothercell. */
        CellI = (LIB_DERIVE_CELL)BListEl (ListCells, i);
        MotherCellI = LibDeriveCellMotherCell (CellI);
        CArea = LibHCellArea (MotherCellI);
        CDepth = 1;

        /* If Bdd of the derive cell is the complement of the Bdd of the*/
        /* current node then an inverter is mandatory so we increase      */

```

gnlmap.c

```
/* the Area of the cell by the area of the best inverter. */
/* Corner case: if the cell is itself the best inverter then we */
/* do not do it. */
if ((MotherCellI != G_BestInverter) &&
    (Bdd != LibDeriveCellBdd (CellI)))
{
    CArea += LibHCellArea (G_BestInverter);
    CDepth++;
}

/* storing the new best depth. */
if ((CDepth < *BestDepth) ||
    ((CDepth == *BestDepth) && (CArea < *BestArea)))
{
    *BestDepth = CDepth;
    *BestArea = CArea;
    *BestCell = CellI;
}
}

}

/*-----*/
/* GnlGetBestNetCell */
/*-----*/
/* This function returns the Best Cell and Best Depth from a given list */
/* cells. The list of cells is a list of LIB_DERIVE_CELL and all have */
/* the same Bdd or !Bdd. A cell having a bdd LibDeriveCellBdd (Cell) */
/* which !Bdd needs also an inverter to realize the !. So the Depth */
/* associated to this cell is its own Depth + inverter Depth. If the */
/* Bdd LibDeriveCellBdd (Cell) of a cell is Bdd then its depth is simply */
/* 1. */
/*-----*/
void GnlGetBestNetCell (ListCells, Bdd, BestCell, BestDepth, BestNet)
BLIST          ListCells;
BDD            Bdd;
LIB_DERIVE_CELL *BestCell;
int             *BestDepth;
int             *BestNet;
{
    int             i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL        MotherCellI;
    int             CDepth;
    int             CNet;
    float           BestArea;
    float           CArea;

    BestArea = MAX_AREA_VALUE;

    *BestDepth = MAX_DEPTH_VALUE;
    *BestCell = NULL;
    *BestNet = MAX_NET_VALUE;

    for (i=0; i<BListSize (ListCells); i++)
```

gnlmap.c

```

{
    /* first pick up the derive cell area thru its mothercell.      */
    CellI = (LIB_DERIVE_CELL)BListElt (ListCells, i);
    MotherCellI = LibDeriveCellMotherCell (CellI);
    CArea = LibHCellArea (MotherCellI);
    CDepth = 1;
    CNet = 1;

    /* If Bdd of the derive cell is the complement of the Bdd of the*/
    /* current node then an inverter is mandatory so we increase     */
    /* the Area of the cell by the area of the best inverter.       */
    /* Corner case: if the cell is itself the best inverter then we */
    /* do not do it.                                                 */
    if ((MotherCellI != G_BestInverter) &&
        (Bdd != LibDeriveCellBdd (CellI)))
    {
        CDepth += 1;
        CNet++;
        CArea += LibHCellArea (G_BestInverter);
    }

    /* storing the new best depth.                                     */
    if ((CNet < *BestNet) ||
        ((CNet == *BestNet) && (CArea < BestArea)))
    {
        *BestNet = CNet;
        *BestDepth = CDepth;
        BestArea = CArea;
        *BestCell = CellI;
    }
}

/*-----
/* GnlCreateSupportNodeFromNode                                         */
/*-----*/
/* This function returns a GNL_SNODE 'SNode' which is constituted of   */
/* all the Sons of 'Node'. All the corresponding GNL_SNODE are chained   */
/* thru 'GnlSNodeNext'.                                                 */
/*-----*/
GNL_STATUS GnlCreateSupportNodeFromNode (Node, SNode)
    GNL_NODE Node;
    GNL_SNODE *SNode;
{
    int i;
    GNL_NODE SonI;
    GNL_SNODE SiNode;
    GNL_SNODE PSNode;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlCreateSNode (&SiNode))
            return (GNL_MEMORY_FULL);
}

```

gnlmap.c

```
SetGnlSNodeNode (SiNode, SonI);
if (i == 0)
{
    *SNode = SiNode;
}
else
    SetGnlSNodeNext (PSNode, SiNode);

PSNode = SiNode;
}

SetGnlSNodeNext (PSNode, NULL);

return (GNL_OK);
}

/*
 * GnlExtractSortedLibVars
 */
/*
 * This function returns a BLIST of GNL_VAR which correspond to the Var
 * associated to each SNode. If the Node of a SNode is a GNL_VARIABLE
 * then its associated var is GnlVarBindVar (GnlNodeSons (Node)). If the
 * Node of a SNode is an operator (GNL_NOT, ...) the its associated var
 * is GnlMapNodeInfoLibVar (Node).
 */
/* Thi slist of vars is then sorted according to their canonical
 * signature.
 */
GNL_STATUS GnlExtractSortedLibVars (SNodes, SortedVars)
    GNL_SNODE      SNodes;
    BLIST      *SortedVars;
{
    GNL_NODE NodeI;
    GNL_VAR  Var;
    GNL_VAR  LibVarI;

    if (BListCreateWithSize (1, SortedVars))
        return (GNL_MEMORY_FULL);

    while (SNodes != NULL)
    {
        NodeI = GnlSNodeNode (SNodes);

        if (GnlNodeOp (NodeI) == GNL_VARIABLE)
        {
            Var = (GNL_VAR) GnlNodeSons (NodeI);
            LibVarI = GnlVarBindVar (Var);
        }
        else if (GnlNodeOp (NodeI) == GNL_CONSTANTE)
        {
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }
        else
        {
            LibVarI = GnlMapNodeInfoLibVar (NodeI);
        }
    }
}
```

gnlmap.c

```
}

/* We add only one time a Var in '*SortedVars'. */
if (!BListMemberOfList (*SortedVars, LibVarI, IntIdentical))
    if (BListAddElt (*SortedVars, (int)LibVarI))
        return (GNL_MEMORY_FULL);

SNodes = GnlSNodeNext (SNodes);
}

/* we sort the inputs in order to respect the ordering of signatures */
qsort (BListAdress (*SortedVars), BListSize (*SortedVars),
       sizeof (GNL_VAR), CmpVarSignature);

return (GNL_OK);
}

/*-----
/* GnlGetSortBindVar
*/
/*-----
/* This function returns the original var associated to 'BindVar' in the*/
/* list 'OriginVars'.
*/
/*-----
GNL_VAR GnlGetSortBindVar (BindVar, SortedVars, OriginVars)
    GNL_VAR BindVar;
    BLIST SortedVars;
    BLIST OriginVars;
{
    int i;

    for (i=0; i<BListSize (SortedVars); i++)
    {
        if (BindVar == (GNL_VAR)BListElt (SortedVars, i))
            break;
    }

    return ((GNL_VAR)BListElt (OriginVars, i));
}

/*-----
/* GnlBindSupportWithLibVars
*/
/*-----
/* This functions associates to each Node of SNode a bind/library var. */
/* Corner case: a Node of a SNode can refer several times to the same */
/* GNL_VARIABLE. We need to associate to this Var always the same Bind */
/* Var.
*/
/*-----
GNL_STATUS GnlBindSupportWithLibVars (SNodes)
    GNL_SNODE SNodes;
{
    int i;
    GNL_VAR VarI;
    GNL_VAR Var;
    GNL_NODE NodeI;
    GNL_VAR_LIB_INFO VarLibInfo;
```

```

/* we use this tag to make sure we create only one single var when */
/* we are in the case where a same GNL_VARIABLE is referenced several*/
/* times. */
SetGnlTag (G_Gnl, GnlTag(G_Gnl)+1);

i = 0;           /* i is the index of the next library variable */
                  /* available. */
while (SNodes != NULL)
{
    if (i > BListSize (G_LibInputs))
    {
        fprintf (stderr,
                  "\n ERROR: Tree not enough decomposed to be mapped\n");
        exit (1);
    }

    NodeI = GnlSNodeNode (SNodes);

    /* If node is a GNL_VARIABLE then we bind a new library */
    /* variable to its var and to the Node GNL_VARIABLE. Do not do*/
    /* it if this variable was already processed. */
    /* variable level ... */
    if (GnlNodeOp (NodeI) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (NodeI);
        if (!GnlVarHook (Var))
        {
            if (GnlVarLibInfoCreate (&VarLibInfo))
                return (GNL_MEMORY_FULL);
            SetGnlVarHook (Var, VarLibInfo);
        }

        /* If the tag is different we did not already processed */
        /* this Variable 'Var'. */
        if (GnlVarTag (Var) != GnlTag (G_Gnl))
        {
            /* we set the tag so in the next pass we know that we */
            /* already visited it. */
            SetGnlVarTag (Var, GnlTag (G_Gnl));
            VarI = (GNL_VAR)BListElt (G_LibInputs, i);
            SetGnlVarBindVar (Var, VarI);
            i++;
        }

        /* We bind also the Node with this variable */
        SetGnlMapNodeInfoLibVar (NodeI, GnlVarBindVar (Var));
    }
    else if (GnlNodeOp (NodeI) == GNL_CONSTANTE)
    {
        SNodes = GnlSNodeNext (SNodes);
        continue;
    }
    else      /* otherwise at the Node level. */
    {
        SetGnlNodeTag (GnlSNodeNode (SNodes), GnlTag (G_Gnl));
        VarI = (GNL_VAR)BListElt (G_LibInputs, i);
    }
}

```

gnlmap.c

```
        SetGnlMapNodeInfoLibVar (GnlSNodeNode (SNodes), VarI);
        i++;
    }

    SNodes = GnlSNodeNext (SNodes);
}

return (GNL_OK);
}

/*-----
/* GnlReassignLibVars
*/
/*-----
/* We redo the library var assignment in order to respect the signatures*/
/* The list 'SortedVars' gives the correct order. */
*/
void GnlReassignLibVars (SNodes, SortedVars)
{
    GNL_SNODE      SNodes;
    BLIST      SortedVars;
{
    GNL_NODE      NodeI;
    GNL_VAR      Var;
    GNL_VAR      BindVar;
    GNL_VAR      NewBindVar;

    while (SNodes != NULL)
    {
        NodeI = GnlSNodeNode (SNodes);

        if (GnlNodeOp (NodeI) == GNL_CONSTANTE)
        {
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }

        BindVar = GnlMapNodeInfoLibVar (NodeI);
        NewBindVar = GnlGetSortBindVar (BindVar, SortedVars, G_LibInputs);

        /* substitute the old bind var 'BindVar' by the correct one which */
        /* is 'NewBindVar'. */
        /* Warning: the substitution can affect also the bind var of Var */
        /* of NodeI in order to be consistant so may be its good to */
        /* update it too. */
        SetGnlMapNodeInfoLibVar (NodeI, NewBindVar);

        SNodes = GnlSNodeNext (SNodes);
    }
}

/*-----
/* GnlIsAlreadyInList
*/
/*-----
/* This functions checks if 'Node' which is a GNL_VARIABLE has already */
/* an equivalent node GNL_VARIABLE which points on the same Var. returns*/
/* if Yes and 0 otherwise. */
*/
```

gnlmap.c

```
/*-----*/
int GnlIsAlreadyInList (Node, NodeSupport)
    GNL_NODE Node;
    BLIST    NodeSupport;
{
    int             i;
    GNL_NODE NodeI;

    for (i=0; i<BListSize (NodeSupport); i++)
    {
        NodeI = (GNL_NODE)BListElt (NodeSupport, i);
        if ((GnlNodeOp (NodeI) == GNL_VARIABLE) &&
            (GnlNodeSons (NodeI) == GnlNodeSons (Node)))
            return (1);
    }

    return (0);
}

/*-----*/
/* GnlSaveInstance */
/*-----*/
/* This function saves for a particular node, its support function and */
/* library variables associated to this support. Both lists are stored */
/* in 'GnlMapNodeInfoNodeSupport' and 'GnlMapNodeInfoNodeSupport' fields */
/* of the Node. */
/*-----*/
GNL_STATUS GnlSaveInstance (Node, SNodes)
    GNL_NODE Node;
    GNL_SNODE     SNodes;
{
    GNL_NODE NodeI;
    GNL_VAR Var;
    GNL_VAR BindVar;
    BLIST    NewList;

    if (GnlMapNodeInfoNodeSupport (Node) == NULL)
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        SetGnlMapNodeInfoNodeSupport (Node, NewList);
    }
    /* resetting the list to store a new instance */          */
    BSize (GnlMapNodeInfoNodeSupport (Node)) = 0;

    if (GnlMapNodeInfoLibVarSupport (Node) == NULL)
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        SetGnlMapNodeInfoLibVarSupport (Node, NewList);
    }
    /* resetting the list to store a new instance */          */
    BSize (GnlMapNodeInfoLibVarSupport (Node)) = 0;
```

gnlmap.c

```
while (SNodes != NULL)
{
    NodeI = GnlSNodeNode (SNodes);

    if (GnlNodeOp (NodeI) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (NodeI);
        /* Already added in the support so not needed to add it      */
        /* one more time.                                              */
        if (GnlIsAlreadyInList (NodeI,
                                GnlMapNodeInfoNodeSupport (Node)))
        {
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }

        /* we add the Node 'NodeI' in the support.                      */
        if (BListAddElt (GnlMapNodeInfoNodeSupport (Node),
                         (int)NodeI))
            return (GNL_MEMORY_FULL);

        /* we add the lib var 'BindVar' in the list of lib vars      */
        BindVar = GnlMapNodeInfoLibVar (NodeI);
        if (BListAddElt (GnlMapNodeInfoLibVarSupport (Node),
                         (int)BindVar))
            return (GNL_MEMORY_FULL);
    }
    else if (GnlNodeOp (NodeI) == GNL_CONSTANTE)
    {
        SNodes = GnlSNodeNext (SNodes);
        continue;
    }
    else
    {
        /* we add the Node 'NodeI' in the support.                      */
        if (BListAddElt (GnlMapNodeInfoNodeSupport (Node),
                         (int)NodeI))
            return (GNL_MEMORY_FULL);

        /* we add the lib var 'BindVar' in the list of lib vars      */
        BindVar = GnlMapNodeInfoLibVar (NodeI);
        if (BListAddElt (GnlMapNodeInfoLibVarSupport (Node),
                         (int)BindVar))
            return (GNL_MEMORY_FULL);
    }

    SNodes = GnlSNodeNext (SNodes);
}

return (GNL_OK);
}

/*-----
/* GnlExpanseSupportNode                                         */
/*-----*/
/* This procedure creates a New SNode 'NewSNodes' from 'SNodes' by   */
/* replacing the SNode at index 'i' by its sons.                      */
/*-----*/
```

gnlmap.c

```
/* Ex: SNodes = [x1 x2 a b x3], i = 2, x2 = a + x4 then 'NewSNodes' is: */
/*      NewSNodes = [x1 a x4 a b x3].                                     */
/*-----*/
void GnlExpanseSupportNode (SNodes, i, NewSNodes, FirstSNode)
{
    GNL_SNODE      SNodes;
    int           i;
    GNL_SNODE      *NewSNodes;
    GNL_SNODE      *FirstSNode;
{
    GNL_SNODE      SNodesC;
    int           j;
    GNL_NODE NodeI;
    GNL_SNODE      PSNode;
    GNL_SNODE      SiNode;
    GNL_NODE SonI;

    *NewSNodes = NULL;

    PSNode = NULL;
    SNodesC = SNodes;
    for (j=0; j<i; j++)
    {
        PSNode = SNodesC;
        SNodesC = GnlSNodeNext (SNodesC);
    }

    NodeI = GnlSNodeNode (SNodesC);

    if (PSNode)
    {
        SetGnlSNodeNext (PSNode, GnlMapNodeInfoSNodes (NodeI));
        *NewSNodes = SNodes;
        *FirstSNode = GnlMapNodeInfoSNodes (NodeI);
        PSNode = GnlSNodeNext (PSNode);
    }
    else
    {
        *NewSNodes = GnlMapNodeInfoSNodes (NodeI);
        *FirstSNode = GnlMapNodeInfoSNodes (NodeI);
        PSNode = *NewSNodes;
    }

    while (GnlSNodeNext (PSNode))
    {
        PSNode = GnlSNodeNext (PSNode);
    }

    SetGnlSNodeNext (PSNode, GnlSNodeNext (SNodesC));
}

/*-----*/
/* GnlDeleteSupportNode                                         */
/*-----*/
/* This procedure does the opposite of 'GnlExpanseSupportNode' and      */
/* de-expanse 'NewSNode' in order to get the previous original support   */
/*-----*/

```

gnlmap.c

```
void GnlDeleteSupportNode (SupportNodes, NewSNode, Index, SNodes, Length)
    GNL_SNODE      SupportNodes;
    GNL_SNODE      NewSNode;
    int           Index;
    GNL_SNODE      SNodes;
    int           Length;
{
    int           j;
    GNL_SNODE      PSNode;

    PSNode = NULL;
    for (j=0; j<Index; j++)
    {
        PSNode = SupportNodes;
        SupportNodes = GnlSNodeNext (SupportNodes);
    }

    if (PSNode == NULL)
    {
        for (j=0; j<Length; j++)
        {
            NewSNode = GnlSNodeNext (NewSNode);
        }
        SetGnlSNodeNext (NewSNode, NULL);
        return;
    }

    for (j=0; j<Length; j++)
    {
        SupportNodes = GnlSNodeNext (SupportNodes);
    }

    SetGnlSNodeNext (PSNode, SNodes);
    SetGnlSNodeNext (SNodes, GnlSNodeNext (SupportNodes));
    SetGnlSNodeNext (SupportNodes, NULL);
}

/*
 * GetRealSizeSupport
 */
/* Returns the real number of different nodes in the support constituted*
 * of SupportNodes and nodes. When NODE variable are used then we count */
/* only one time this variable. */
int GetRealSizeSupport (SupportNodes, Sons)
    GNL_SNODE      SupportNodes;
    BLIST      Sons;
{
    int           Size;
    GNL_NODE Node;
    GNL_NODE SonJ;
    GNL_VAR   Var;
    int           j;

    if (BListSize (Sons) == 1)
```

gnlmap.c

```

    return (1);

    Size = BListSize (Sons);
    while (SupportNodes)
    {
        Node = GnlSNodeNode (SupportNodes);
        if (GnlNodeOp (Node) == GNL_VARIABLE)
        {
            Var = (GNL_VAR) GnlNodeSons (Node);
            for (j=0; j<BListSize (Sons); j++)
            {
                SonJ = (GNL_NODE) BListElt (Sons, j);
                if ((GnlNodeOp (SonJ) == GNL_VARIABLE) &&
                    (Var == (GNL_VAR) GnlNodeSons (SonJ)))
                    Size--;
            }
        }
        Size++;
        SupportNodes = GnlSNodeNext (SupportNodes);
    }

    return (Size-1);
}

/*
 * GnlGetBestAreaCellOnNode
 */
/*
 * This procedure is the main Mapping Area function. It looks for all
 * the cells which can realize the functionality starting from GNL_NODE
 * 'Node' and which leaves are the ones in 'SupportNodes'. It select
 * the best cell which involves the best overall area mapping. Recursive
 * call is performed on other functionalities starting from Node (which
 * are still defined by the support 'NewSupportNodes'. Indirect
 * recursive call is also performed on 'GnlBestAreaMapNode' when a cell
 * has been found. This call is performed on all the Node of the current
 * support. The best area is stored on the field GnlMapNodeInfoBestArea
 * of 'Node'.
 * 'Done' is 1 if the mapper was able to map and '0' otherwise.
 */
GNL_STATUS GnlGetBestAreaCellOnNode (Node, SupportNodes, Index,
                                      SizeSupportNodes, FirstSNode, Done)
{
    GNL_NODE      Node;
    GNL_SNODE     SupportNodes;
    int           SizeSupportNodes;
    int           Index;
    GNL_SNODE     FirstSNode;
    int           *Done;

    int           i;
    GNL_VAR       VarI;
    GNL_SNODE     SNodes;
    int           CellFound;
    BDD_PTR       NewBddPtr;
    GNL_STATUS    Status;
    BDD           NewBdd;
    GNL_NODE      NodeI;
}

```

gnlmap.c

```
GNL_SNODE           NewSupportNodes;
GNL_SNODE           SNodei;
int                SizeNewSupport;
int                CDone;
BLIST              SortedVars;
float              Area;
float              NArea;
LIB_DERIVE_CELL    BestCell;
int                NDepth;
int                Depth;
int                MaxDepth;
int                RealSizeSupport;

#ifndef STAT
    G_NB_TARGET_FUNC++;
#endif

*Done = 0;

if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
{
    *Done = 1;
    return (GNL_OK);
}

SNodes = FirstSNode;
i = Index;
while (SNodes != NULL)
{
    NodeI = GnlSNodeNode (SNodes);

    /* It is not necessary to expanse Nodes with index less than      */
    /* 'Index' because it has been already done.                      */
    if ((GnlNodeOp (NodeI) != GNL_VARIABLE) &&
        (GnlNodeOp (NodeI) != GNL_CONSTANTE))
    {

#ifndef ENHANCE_1
    RealSizeSupport = GetRealSizeSupport (SupportNodes,
                                         GnlNodeSons (NodeI));
    if (RealSizeSupport > G_MaxCellSizeSupport)
    {
        i++;
        SNodes = GnlSNodeNext (SNodes);
        continue;
    }

    SizeNewSupport = SizeSupportNodes +
                    BListSize (GnlNodeSons (NodeI))-1;
#else
    SizeNewSupport = SizeSupportNodes +
                    BListSize (GnlNodeSons (NodeI))-1;

    if (SizeNewSupport > G_MaxCellSizeSupport)
    {
        i++;
    }

```

gnlmap.c

```
    SNodes = GnlSNodeNext (SNodes);
    continue;
}
#endif

    GnlExpanseSupportNode (SupportNodes, i, &NewSupportNodes,
                           &SNodei);

    if (GnlGetBestAreaCellOnNode (Node, NewSupportNodes, i,
                                 SizeNewSupport, SNodei, &CDone))
        return (GNL_MEMORY_FULL);

    *Done |= CDone;

    GnlDeleteSupportNode (SupportNodes, NewSupportNodes, i,
                          SNodes,
                          SizeNewSupport-SizeSupportNodes);

}

i++;
SNodes = GnlSNodeNext (SNodes);
}

/* We bind the Nodes of the support to bdd primary inputs of the tech*/
/* library. */
if (GnlBindSupportWithLibVars (SupportNodes))
    return (GNL_MEMORY_FULL);

if ((Status = GnlBuildDirectBddOnMapNode (Node)))
    return (Status);

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);

NewBddPtr = GetBddPtrFromBdd (NewBdd);

/* The Bdd is simply a constant. */
if ((NewBdd == bdd_one ()) || (NewBdd == bdd_zero ()))
{
    *Done = 1;
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    SetGnlMapNodeInfoBestArea (Node, 0.0);
    SetGnlMapNodeInfoBestDepth (Node, 0);
    return (GNL_OK);
}

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));
/* If no cell found we simply return. */
if (!CellFound)
{
    return (GNL_OK);
```

gnlmap.c

```
}

#endif STAT
    G_NB_CELL_FOUND++;
#endif

*Done = 1;

/* Computing the best Area taking this cell as solution */
GnlGetBestAreaCell (LibBddInfoDeriveCells (NewBddPtr),
                    NewBdd, &BestCell, &Area, &Depth);

MaxDepth = 0;
SNodes = SupportNodes;
while (SNodes != NULL)
{
    if (GnlBestAreaMapNode (GnlSNodeNode (SNodes), &NArea,
                           &NDepth, Done))
        return (GNL_MEMORY_FULL);

    /* Impossible to map 'GnlSNodeNode (SNodes)'.
    if (!(*Done))
        return (GNL_OK);

    Area += NArea;

    if (NDepth > MaxDepth)
        MaxDepth = NDepth;

    /* We cope since we have already found a better solution */
    if (Area > GnlMapNodeInfoBestArea (Node))
        break;
    SNodes = GnlSNodeNext (SNodes);
}

/* If better then save everything: Area, Bdd, vars associations */
if ((Area < GnlMapNodeInfoBestArea (Node)) ||
    ((Area == GnlMapNodeInfoBestArea (Node)) &&
     (MaxDepth+Depth < GnlMapNodeInfoBestDepth (Node))))
{
    SetGnlMapNodeInfoBestArea (Node, Area);
    SetGnlMapNodeInfoBestDepth (Node, MaxDepth+Depth);
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    if (GnlSaveInstance (Node, SupportNodes))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlBestAreaMapNode
/*-----
/* This is the main procedure for the Mapping area option. It looks for */
/* the best mapping starting from node 'Node' and returns the best Area */
```

gnlmap.c

```
/* seen so far. The area information is also stored in the field */
/* 'GnlMapNodeInfoBestArea' of node 'Node' in oreder to pick it up if */
/* we recall this function on the same Node. */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise. */
/*-----*/
GNL_STATUS GnlBestAreaMapNode (Node, Area, Depth, Done)
    GNL_NODE Node;
    float *Area;
    int *Depth;
    int *Done;
{
    GNL_SNODE SupportNode;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        *Done = 1;
        *Area = 0.0;
        *Depth = 0;
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        *Done = 1;
        *Area = 0.0;
        *Depth = 0;
        return (GNL_OK);
    }

    if ((GnlMapNodeInfoBestArea (Node) != 0.0) &&
        (GnlMapNodeInfoBestArea (Node) != MAX_AREA_VALUE))
    {
        *Done = 1;
        *Area = GnlMapNodeInfoBestArea (Node);
        *Depth = GnlMapNodeInfoBestDepth (Node);
        return (GNL_OK);
    }

SetGnlMapNodeInfoBestArea (Node, MAX_AREA_VALUE);
SetGnlMapNodeInfoBestDepth (Node, MAX_DEPTH_VALUE);

SupportNode = GnlMapNodeInfoSNodes (Node);

if (GnlGetBestAreaCellOnNode (Node, SupportNode, 0,
                            BListSize (GnlNodeSons (Node)),
                            SupportNode, Done))
    return (GNL_MEMORY_FULL);

/* we were not able to map everything. */
if ((*Done) == 0)
{
    return (GNL_OK);
}

*Area = GnlMapNodeInfoBestArea (Node);
```

gnlmap.c

```
*Depth = GnlMapNodeInfoBestDepth (Node);

    return (GNL_OK);
}

/*
/* GnlGetCapOnNode
*/
/*
/* This function extracts the capacitance of the input pin associated to*/
/* SNode 'SNodes'.
*/
/*
float GnlGetCapOnNode (SNodes, Cell)
    GNL_SNODE      SNodes;
    LIB_DERIVE_CELL   Cell;
{
    GNL_NODE      Node;
    GNL_VAR       BindVar;
    LIB_CELL      MotherCell;
    GNL_VAR       Var;
    LIBC_PIN      Pins;
    LIBC_NAME_LIST ListPinName;
    LIBC_CELL      LibCell;

    Node = GnlSNodeNode (SNodes);
    BindVar = GnlMapNodeInfoLibVar (Node);

    MotherCell = LibDeriveCellMotherCell (Cell);

    Var = GnlGetOriginalVar (Cell, MotherCell, BindVar);
    if (!Var)
        return (0.0);

    LibCell = LibHCellLibcCell (MotherCell);
    Pins = LibCellPins (LibCell);
    for (;Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins);
        if (!strcmp (GnlVarName (Var), LibNameListName (ListPinName)))
        {
            if (LibPinCapa (Pins) != 0.0)
                return (LibPinCapa (Pins));
            return (LibDefaultInputPinCap (G_GnlLibc));
        }
    }

    return (LibDefaultInputPinCap (G_GnlLibc));
}

/*
/* GnlGetAssociatedPin
*/
/*
/* This function returns the associated pin of the cell 'Cell' which is */
/* associated to the SNODE 'SNodes'.
*/
/*
LIBC_PIN GnlGetAssociatedPin (SNodes, Cell)
    GNL_SNODE      SNodes;
```

gnlmap.c

```
LIB_DERIVE_CELL      Cell;
{
    GNL_NODE          Node;
    GNL_VAR           BindVar;
    LIB_CELL          MotherCell;
    GNL_VAR           Var;
    LIBC_PIN          Pins;
    LIBC_NAME_LIST    ListPinName;
    LIBC_CELL          LibCell;

    Node = GnlSNodeNode (SNodes);
    BindVar = GnlMapNodeInfoLibVar (Node);

    MotherCell = LibDeriveCellMotherCell (Cell);

    Var = GnlGetOriginalVar (Cell, MotherCell, BindVar);
    if (!Var)
        return (NULL);

    LibCell = LibHCellLibcCell (MotherCell);
    Pins = LibCellPins (LibCell);
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins);
        if (!strcmp (GnlVarName (Var), LibNameListName (ListPinName)))
        {
            return (Pins);
        }
    }

    return (NULL);
}

/*-----
/* GnlGetCapaFromPin
/*-----*/
/* Returns the associated capacitance of pin 'Pin'. */
/*-----*/
float GnlGetCapaFromPin (Pin)
    LIBC_PIN Pin;
{
    if (LibPinCapa (Pin) != 0.0)
        return (LibPinCapa (Pin));

    return (LibDefaultInputPinCap (G_GnlLibc));
}

/*-----
/* GnlGetBestPowerCellOnNode
/*-----*/
/* This procedure is the main Mapping Power function. It looks for all */
/* the cells which can realize the functionality starting from GNL_NODE */
/* 'Node' and which leaves are the ones in 'SupportNodes'. It select */
/* the best cell which involves the best overall Power mapping.Recursive*/
/* call is performed on other functionalities starting from Node (which */
/* are still defined by the support 'NewSupportNodes'). Indirect */
*/
```

gnlmap.c

```
/* recursive call is also performed on 'GnlBestAreaMapNode' when a cell */
/* has been found. This call is perfomed on all the Node of the current */
/* support.The best Power is stored on the field GnlMapNodeInfoBestPower*/
/* of 'Node'. */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise. */
/*-----*/
GNL_STATUS GnlGetBestPowerCellOnNode (Node, Capacitance, SupportNodes,
                                         Index, SizeSupportNodes, Rec, Done)
{
    GNL_NODE      Node;
    float        Capacitance;
    GNL_SNODE     SupportNodes;
    int          SizeSupportNodes;
    int          Index;
    int          Rec;
    int          *Done;
{
    int           i;
    GNL_VAR       VarI;
    GNL_SNODE     SNodes;
    int          CellFound;
    BDD_PTR       NewBddPtr;
    GNL_STATUS    Status;
    BDD          NewBdd;
    GNL_NODE      NodeI;
    GNL_SNODE     NewSupportNodes;
    GNL_SNODE     SNodei;
    int          SizeNewSupport;
    int          CDone;
    BLIST         SortedVars;
    float         Power;
    float         NPower;
    LIB_DERIVE_CELL BestCell;
    int          NDepth;
    int          Depth;
    int          MaxDepth;
    float         CapaOnNode;

#ifdef STAT
    G_NB_TARGET_FUNC++;
#endif

    *Done = 0;
    if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
        (GnlNodeOp (Node) == GNL_CONSTANTE))
    {
        *Done = 1;
        return (GNL_OK);
    }

    SNodes = SupportNodes;
    i = 0;
    while (SNodes != NULL)
    {
        NodeI = GnlSNodeNode (SNodes);

        /* It is not necesary to expanse Nodes with index less than */

```

gnlmap.c

```
/* 'Index' because it has been already done. */  
if ((i >= Index) &&  
    (GnlNodeOp (NodeI) != GNL_VARIABLE) &&  
    (GnlNodeOp (NodeI) != GNL_CONSTANTE))  
{  
    SizeNewSupport = SizeSupportNodes +  
        BListSize (GnlNodeSons (NodeI))-1;  
  
    if (SizeNewSupport > G_MaxCellSizeSupport)  
    {  
        i++;  
        SNodes = GnlSNodeNext (SNodes);  
        continue;  
    }  
  
    GnlExpanseSupportNode (SupportNodes, i, &NewSupportNodes,  
                           &SNodei);  
  
    if (GnlGetBestPowerCellOnNode (Node, Capacitance,  
                                  NewSupportNodes, i,  
                                  SizeNewSupport, Rec+1, &CDone))  
        return (GNL_MEMORY_FULL);  
  
    *Done |= CDone;  
  
    GnlDeleteSupportNode (SupportNodes, NewSupportNodes, i,  
                          SNodes,  
                          SizeNewSupport-SizeSupportNodes);  
  
    if (!G_Effort && CDone)  
        break;  
}  
  
i++;  
SNodes = GnlSNodeNext (SNodes);  
}  
  
/* We bind the Nodes of the support to bdd primary inputs of the tech*/  
/* library. */  
if (GnlBindSupportWithLibVars (SupportNodes))  
    return (GNL_MEMORY_FULL);  
  
if ((Status = GnlBuildDirectBddOnMapNode (Node)))  
    return (Status);  
  
#ifdef STAT  
    G_ROBDD_BUILT++;  
#endif  
  
NewBdd = GnlMapNodeInfoBdd (Node);  
  
NewBddPtr = GetBddPtrFromBdd (NewBdd);  
  
/* The Bdd is simply a constant. */  
if ((NewBdd == bdd_one ()) || (NewBdd == bdd_zero ()))  
{  
    *Done = 1;
```

gnlmap.c

```
SetGnlMapNodeInfoBestBdd (Node, NewBdd) ;
SetGnlMapNodeInfoBestPower (Node, 0.0) ;
SetGnlMapNodeInfoBestDepth (Node, 0) ;
return (GNL_OK) ;
}

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr))) ;

/* If no cell found we rebuild the Bdd with a pseudo canonical order */
if (!CellFound)
{
    /* We compute the signature of the Node tree from the previous      */
    /* computed Bdd.                                                       */
    if (GnlComputeVarsSignatureFromBdd (G_LibGnl, NewBdd))           */
        return (GNL_MEMORY_FULL) ;

    if (GnlExtractSortedLibVars (SupportNodes, &SortedVars))          */
        return (GNL_MEMORY_FULL) ;

    GnlReassignLibVars (SupportNodes, SortedVars) ;

    if ((Status = GnlBuildDirectBddOnMapNode (Node)))
        return (Status) ;

#endif STAT
G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node) ;
NewBddPtr = GetBddPtrFromBdd (NewBdd) ;

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr))) ;

BListQuickDelete (&SortedVars) ;
}

if (CellFound)
{
#endif STAT
G_NB_CELL_FOUND++;
#endif

*Done = 1;

/* Computing the best Power taking this cell as solution                  */
GnlGetBestPowerCell (LibBddInfoDeriveCells (NewBddPtr),
                     Capacitance, NewBdd, &BestCell, &Power,
                     &Depth) ;

MaxDepth = 0;
SNodes = SupportNodes;
while (SNodes != NULL)
{
    /* we extract the capacitance of the pin associated to      */
    /* SNodes.                                                       */
    /* */
```

```

CapaOnNode = GnlGetCapOnNode (SNodes, BestCell);

if (GnlBestPowerMapNode (GnlSNodeNode (SNodes), CapaOnNode,
                        &NPower, &NDepth, Rec, Done))
    return (GNL_MEMORY_FULL);

/* Impossible to map 'GnlSNodeNode (SNodes)' */
if (!(*Done))
    return (GNL_OK);

Power += NPower;

if (NDepth > MaxDepth)
    MaxDepth = NDepth;

/* We cope since we have already found a better solution */
if (Power > GnlMapNodeInfoBestPower (Node))
    break;
SNodes = GnlSNodeNext (SNodes);
}

/* If better then save everything: Power, Bdd, vars associations */
if ((Power < GnlMapNodeInfoBestPower (Node)) ||
    (Power == GnlMapNodeInfoBestPower (Node)) &&
    (MaxDepth+Depth < GnlMapNodeInfoBestDepth (Node)))
{
    SetGnlMapNodeInfoBestPower (Node, Power);
    SetGnlMapNodeInfoBestDepth (Node, MaxDepth+Depth);
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    if (GnlSaveInstance (Node, SupportNodes))
        return (GNL_MEMORY_FULL);
}
}

return (GNL_OK);
}

/*
 * GnlBestPowerMapNode
 */
/*
 * This is the main procedure for the Mapping Power option. It looks for*
 * the best mapping starting from node 'Node' and returns the best Power*
 * seen so far. The Power information is also stored in the field      */
/* 'GnlMapNodeInfoBestPower' of node 'Node' in order to pick it up if   */
/* we recall this function on the same Node.                            */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise.       */
*/
GNL_STATUS GnlBestPowerMapNode (Node, Capacitance, Power, Depth, Rec,
                               Done)
{
    GNL_NODE      Node;
    float        Capacitance;
    float        *Power;
    int          *Depth;
    int          Rec;
    int          *Done;
}

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

gnlmap.c

```
{  
    GNL_SNODE      SupportNode;  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        *Done = 1;  
        *Power = 0.0;  
        *Depth = 0;  
        return (GNL_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
    {  
        *Done = 1;  
        *Power = 0.0;  
        *Depth = 0;  
        return (GNL_OK);  
    }  
  
    if ((GnlMapNodeInfoBestPower (Node) != 0.0) &&  
        (GnlMapNodeInfoBestPower (Node) != MAX_POWER_VALUE))  
    {  
        *Done = 1;  
        *Power = GnlMapNodeInfoBestPower (Node);  
        *Depth = GnlMapNodeInfoBestDepth (Node);  
        return (GNL_OK);  
    }  
  
    SetGnlMapNodeInfoBestPower (Node, MAX_POWER_VALUE);  
    SetGnlMapNodeInfoBestDepth (Node, MAX_DEPTH_VALUE);  
  
    SupportNode = GnlMapNodeInfoSNodes (Node);  
  
    if (GnlGetBestPowerCellOnNode (Node, Capacitance, SupportNode, 0,  
                                  BListSize (GnlNodeSons (Node)),  
                                  Rec, Done))  
        return (GNL_MEMORY_FULL);  
  
    /* we were not able to map everything. */  
    if ((*Done) == 0)  
    {  
        return (GNL_OK);  
    }  
  
    *Power = GnlMapNodeInfoBestPower (Node);  
    *Depth = GnlMapNodeInfoBestDepth (Node);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGetFirstOutputPinCell */  
/*-----*/  
/* returns the first output in the derive cell 'BestCell' */  
/*-----*/
```

gnlmap.c

```
LIBC_PIN GnlGetFirstOutputPinCell (BestCell)
    LIB_DERIVE_CELL      BestCell;
{
    LIB_CELL            MotherCell;
    LIBC_CELL           LibCell;
    LIBC_PIN            Pins;

    MotherCell = LibDeriveCellMotherCell (BestCell);
    LibCell = LibHCellLibcCell (MotherCell);

    Pins = LibCellPins (LibCell);

    /* we look for the first output pin. */
    for (;Pins != NULL; Pins = LibPinNext(Pins)) {
        if (LibPinDirection (Pins) == OUTPUT_E)
            break;
    }

    return (Pins);
}

/*-----
 * GnlComputeNLCellDelay
 *-----
 * Returns the delay of the cell 'BestCell' having an output capacitance*
 * of 'Capacitance' corresponding to the propagation from pin 'Pin' on  *
 * which there is a transition delay of 'Transition'.                  */
void GnlComputeNLCellDelay (BestCell, OutPin, InPin, Capacitance,
                           TransitionR, TransitionF, DelayR, DelayF,
                           TransR, TransF)
    LIB_DERIVE_CELL      BestCell;
    LIBC_PIN             OutPin;
    LIBC_PIN             InPin;
    float                Capacitance;
    float                TransitionR;
    float                TransitionF;
    float                *DelayR;
    float                *DelayF;
    float                *TransR;
    float                *TransF;
{
    LIB_CELL            MotherCell;
    LIBC_CELL           LibCell;
    LIBC_PIN            Pins;

    MotherCell = LibDeriveCellMotherCell (BestCell);
    LibCell = LibHCellLibcCell (MotherCell);

    LibDelayArcCell (TransitionR, TransitionF, Capacitance, 0.0, 1,
                     LibCell, InPin, OutPin, (float*)DelayR, (float*)TransR,
                     (float*)DelayF, (float*)TransF);
}
```

gnlmap.c

```
/*-----*/
/* GnlGetBestNLDelayCell */
/*-----*/
void GnlGetBestNLDelayCell (Node, ListCells, Capacitance, Bdd, BestCell,
                           OutPin)
    GNL_NODE      Node;
    BLIST         ListCells;
    float         Capacitance;
    BDD          Bdd;
    LIB_DERIVE_CELL *BestCell;
    LIBC_PIN     *OutPin;
{
    int           i;
    LIB_DERIVE_CELL CellI;
    LIB_CELL      MotherCellI;
    float         BestNLDelay;
    LIBC_CELL     LibCell;
    LIBC_PIN      Pins;
    LIBC_PIN      OPin;
    float         MaxCelldelay;
    float         MinMaxCellDelay;
    float         PinCapa;
    float         DelayR;
    float         DelayF;
    float         TransR;
    float         TransF;
    char          *CompoName;

    *BestCell = NULL;
    CompoName = NULL;

    if (GnlMapNodeInfoOriginalCompo (Node))
        CompoName = GnlUserComponentName (
            (GNL_USER_COMPONENT)GnlMapNodeInfoOriginalCompo (Node));
    MinMaxCellDelay = MAX_NL_DELAY_VALUE;

    for (i=0; i<BListSize (ListCells); i++)
    {
        /* first pick up the derive cell area thru its mothercell. */
        CellI = (LIB_DERIVE_CELL)BListElt (ListCells, i);
        MotherCellI = LibDeriveCellMotherCell (CellI);

        /* we found the same gate as the kept one. So we retake this */
        /* one. */
        if (CompoName)
        {
            if (!strcmp (CompoName, LibHCellName (MotherCellI)))
            {
                *OutPin = GnlGetFirstOutputPinCell (CellI);
                *BestCell = CellI;
                return;
            }
        }
        else
        {
            *BestCell = NULL;
        }
    }
}
```

```

    }

/* If Bdd of the derive cell is the complement of the Bdd of the*/
/* current node then an inverter is mandatory so we increase      */
/* the Area of the cell by the area of the best inverter.          */
/* Corner case: if the cell is itself the best inverter then we */
/* do not do it.                                                 */
if ((MotherCellI != G_BestInverter) &
    (Bdd != LibDeriveCellBdd (CellI)))
{
    continue;
}

OPin = GnlGetFirstOutputPinCell (CellI);

if (!OPin)
{
    fprintf (stderr,
              " ERROR: cannot find output pin of cell '%s'\n",
              LibCellName (LibCell));
    exit (1);
}

LibCell = LibHCellLibcCell (MotherCellI);
Pins = LibCellPins (LibCell);

/* we compute the max cell delay with an output capacitance      */
/* 'Capacitance' and an input transition of 0.0.                  */
MaxCelldelay = 0.0;
for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    if (LibPinDirection (Pins) != INPUT_E)
        continue;

    PinCapa = GnlGetCapaFromPin (Pins);

    /* we suppose all the pins to have a Fall/Rise Transition   */
    /* equals to 0.0                                              */
    GnlComputeNLCellDelay (CellI, OPin, Pins, Capacitance,
                           0.0, 0.0,
                           &DelayR, &DelayF, &TransR, &TransF);
    if (DelayR > MaxCelldelay)
        MaxCelldelay = DelayR;
    if (DelayF > MaxCelldelay)
        MaxCelldelay = DelayF;
}

if (MinMaxCellDelay > MaxCelldelay)
{
    *BestCell = CellI;
    *OutPin = OPin;
    MinMaxCellDelay = MaxCelldelay;
}
}

}

```

gnlmap.c

```
/*-----*/
/* PrintNodeAndMatchCell */
/*-----*/
void PrintNodeAndMatchCell (Node, BestCell)
    GNL_NODE      Node;
    LIB_DERIVE_CELL  BestCell;
{
    LIB_CELL      MotherCell;
    LIBC_CELL     LibCell;

    MotherCell = LibDeriveCellMotherCell (BestCell);
    LibCell = LibHCellLibcCell (MotherCell);

    fprintf (stderr, "\n");
    fprintf (stderr, " MATCH CELL = %s\n", LibCellName (LibCell));
    PrintNodeRec (Node, 0);
    fprintf (stderr, "\n");
}

/*-----*/
/* GetNbNetOccurence */
/*-----*/
int GetNbNetOccurence (SupportNodes, SNodes)
    GNL_SNODE      SupportNodes;
    GNL_SNODE      SNodes;
{
    GNL_NODE      Node;
    GNL_VAR       Var;
    GNL_NODE      NodeI;
    GNL_VAR       VarI;
    int           NbOccur;

    Node = GnlSNodeNode (SNodes);
    if (GnlNodeOp (Node) != GNL_VARIABLE)
        return (1);
    Var = (GNL_VAR) GnlNodeSons (Node);

    /* May be node VARIABLE used several times. */
    NbOccur = 0;
    while (SupportNodes != NULL)
    {
        NodeI = GnlSNodeNode (SupportNodes);
        if (GnlNodeOp (Node) != GNL_VARIABLE)
        {
            SupportNodes = GnlSNodeNext (SupportNodes);
            continue;
        }

        VarI = (GNL_VAR) GnlNodeSons (NodeI);

        if (Var == VarI)
            NbOccur++;

        SupportNodes = GnlSNodeNext (SupportNodes);
    }
}
```

gnlmap.c

```
    }

    return (NbOccur);
}

/*-----
/* GnlGetBestNLDelayCellOnNode
*/
/*-----
/* This procedure is the main Non linera delay Mapping. It looks for all*/
/* the cells which can realize the functionality starting from GNL_NODE */
/* 'Node' and which leaves are the ones in 'SupportNodes'. It select */
/* the best cell which involves the best overall NL-Delay mapping. */
/* Recursive call is performed on other functionalities starting from */
/* Node (which are still defined by the support 'NewSupportNodes'). */
/* Indirect recursive call is also performed on 'GnlBestNLDelayMapNode' */
/* when a cell has been found. This call is perfomed on all the Node of */
/* the current support.The best NL-delay is stored on the field */
/* 'GnlMapNodeInfoBestArrival{r,F}' of 'Node'. */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise. */
/*-----*/
GNL_STATUS GnlGetBestNLDelayCellOnNode (Node, Capacitance, SupportNodes,
                                         Index, SizeSupportNodes,
                                         FirstSNode, Done)

    GNL_NODE      Node;
    float         Capacitance;
    GNL_SNODE     SupportNodes;
    int           Index;
    int           SizeSupportNodes;
    GNL_SNODE     FirstSNode;
    int           *Done;

{
    int           i;
    GNL_VAR       VarI;
    GNL_SNODE     SNodes;
    int           CellFound;
    BDD_PTR       NewBddPtr;
    GNL_STATUS    Status;
    BDD           NewBdd;
    GNL_NODE      NodeI;
    GNL_SNODE     NewSupportNodes;
    GNL_SNODE     SNodei;
    int           SizeNewSupport;
    int           CDone;
    BLIST          SortedVars;
    float          MaxNLDelay;
    float          MaxCurrentNLDelay;
    float          NLDelayR;
    float          NLDelayF;
    float          NArrivalR;
    float          NArrivalF;
    float          DelayR;
    float          DelayF;
    float          TransR;
    float          TransF;
    float          CellDelay;
    float          NTransitionR;
    float          NTransitionF;
```

gnlmap.c

```
float          MaxNTransitionR;
float          MaxNTransitionF;
float          CriticPinTransition;
LIB_DERIVE_CELL BestCell;
float          MaxNLDelayR;
float          MaxNLDelayF;
float          CapaOnNode;
LIBC_PIN      Pin;
LIBC_PIN      OutPin;
int           RealSizeSupport;
LIBC_TIMING   Timing;
float          MaxArrival;
float          WireEstim;
int           NetOccur;

#define STAT
G_NB_TARGET_FUNC++;
#endif

*Done = 0;

if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
{
    *Done = 1;
    return (GNL_OK);
}

SNodes = FirstSNode;
i = Index;
while (SNodes != NULL)
{
    NodeI = GnlSNodeNode (SNodes);

    if ((GnlNodeOp (NodeI) != GNL_VARIABLE) &&
        (GnlNodeOp (NodeI) != GNL_CONSTANTE))
    {
        RealSizeSupport = GetRealSizeSupport (SupportNodes,
                                              GnlNodeSons (NodeI));
        if (RealSizeSupport > G_MaxCellSizeSupport)
        {
            i++;
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }

        SizeNewSupport = SizeSupportNodes +
                        BListSize (GnlNodeSons (NodeI))-1;

        GnlExpanseSupportNode (SupportNodes, i, &NewSupportNodes,
                               &SNodei);

        if (GnlGetBestNLDelayCellOnNode (Node, Capacitance,
                                         NewSupportNodes, i,
                                         SizeNewSupport, SNodei,
                                         &CDone))
    }
}
```

```

        return (GNL_MEMORY_FULL);

    *Done |= CDone;

    GnlDeleteSupportNode (SupportNodes, NewSupportNodes, i,
                          SNodes,
                          SizeNewSupport-SizeSupportNodes);
}

i++;
SNodes = GnlSNodeNext (SNodes);
}

/* We bind the Nodes of the support to bdd primary inputs of the tech*/
/* library. */
if (GnlBindSupportWithLibVars (SupportNodes))
    return (GNL_MEMORY_FULL);

if ((Status = GnlBuildDirectBddOnMapNode (Node)))
    return (Status);

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);

NewBddPtr = GetBddPtrFromBdd (NewBdd);

/* The Bdd is simply a constant. */
if ((NewBdd == bdd_one ()) || (NewBdd == bdd_zero ()))
{
    *Done = 1;
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    SetGnlMapNodeInfoBestArrivalR (Node, 0.0);
    SetGnlMapNodeInfoBestArrivalF (Node, 0.0);
    SetGnlMapNodeInfoTransitionR (Node, 0.0);
    SetGnlMapNodeInfoTransitionF (Node, 0.0);
    return (GNL_OK);
}

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));
/* If no cell found there is no match. */
if (!CellFound)
    return (GNL_OK);

#ifndef STAT
    G_NB_CELL_FOUND++;
#endif

*Done = 1;

/* Selecting the best gate minimizing the NL-delay. */
GnlGetBestNLDelayCell (Node, LibBddInfoDeriveCells (NewBddPtr),

```

```

Capacitance, NewBdd, &BestCell, &OutPin);

if (!BestCell)
    return (GNL_OK);

/*
PrintNodeAndMatchCell (Node, BestCell);
*/

/* we scan all the pins of the selected gate and call recursively */
/* the mapping on the associated nodes. Then we compute the max */
/* arrival rise and fall at the output of the gate. */
MaxNLDelayR = 0.0;
MaxNLDelayF = 0.0;
MaxNLDelay = 0.0;
/* We extract the max delay already computed on node 'Node'. */
MaxCurrentNLDelay = (GnlMapNodeInfoBestArrivalR (Node) <
                      GnlMapNodeInfoBestArrivalF (Node) ?
                      GnlMapNodeInfoBestArrivalF (Node) :
                      GnlMapNodeInfoBestArrivalR (Node));
MaxNTransitionR = 0.0;
MaxNTransitionF = 0.0;
SNodes = SupportNodes;
while (SNodes != NULL)
{
    Pin = GnlGetAssociatedPin (SNodes, BestCell);

/*
    NetOccur = GetNbNetOccurence (SupportNodes, SNodes);
*/

    if (!Pin)
    {
        SNodes = GnlSNodeNext (SNodes);
        continue;
    }

    /* we extract the capacitance of the pin associated to */
    /* 'SNodes' and add the default wire capacitance. */
/*
WireEstim = G_WireCapa/NetOccur;
*/
    WireEstim = G_WireCapa;
    CapaOnNode = GnlGetCapaFromPin (Pin) + WireEstim;

    /* We compute recursively the mapping on the associated node */
    if (GnlBestNLDelayMapNode (GnlSNodeNode (SNodes), CapaOnNode,
                               &NArrivalR, &NArrivalF,
                               &NTransitionR, &NTransitionF,
                               Done))
        return (GNL_MEMORY_FULL);

    /* Computing the delay of the cell. */
    GnlComputeNLCellDelay (BestCell, OutPin, Pin, Capacitance,
                          NTransitionR, NTransitionF,
                          &DelayR, &DelayF, &TransR, &TransF);
}

```

```

/* The total Arrival time at the output of the cell is the one*/
/* composed of: */                                     */
/*           - cell delay                                */
/*           - arrival time of the inputs               */
/*           - (wire delay) (to implement !)             */
Timing = LibGetArcTiming (Pin, OutPin);

if (!Timing)
{
    MaxArrival = (NArrivalR < NArrivalF ? NArrivalF : NArrivalR);
    NLDelayR = DelayR + MaxArrival;
    NLDelayF = DelayF + MaxArrival;
}
else
{
    switch (LibTimingSense (Timing)) {
        case POSITIVE_UNATE_E:
            NLDelayR = DelayR + NArrivalR;
            NLDelayF = DelayF + NArrivalF;
            break;

        case NEGATIVE_UNATE_E:
            NLDelayR = DelayR + NArrivalF;
            NLDelayF = DelayF + NArrivalR;
            break;

        case NON_UNATE_E:
            MaxArrival = (NArrivalR < NArrivalF ? NArrivalF :
                           NArrivalR);
            NLDelayR = DelayR + MaxArrival;
            NLDelayF = DelayF + MaxArrival;
            break;
    }
}

/* Impossible to map 'GnlSNodeNode (SNodes)' */          */
if (!(*Done))
    return (GNL_OK);

if (NLDelayR > MaxNLDelayR)
{
    MaxNLDelayR = NLDelayR;
    if (MaxNLDelayR > MaxNLDelay)
        MaxNLDelay = MaxNLDelayR;
}

if (NLDelayF > MaxNLDelayF)
{
    MaxNLDelayF = NLDelayF;
    if (MaxNLDelayF > MaxNLDelay)
        MaxNLDelay = MaxNLDelayF;
}

if (TransR > MaxNTransitionR)
    MaxNTransitionR = TransR;

if (TransF > MaxNTransitionF)

```

gnlmap.c

```
        MaxNTransitionF = TransF;

        /* We cope since we have already found a better solution      */
        if (MaxNLDelay >= MaxCurrentNLDelay)
            break;

        SNodes = GnlSNodeNext (SNodes);
    }

/* If better then save everything: NL-delay, Bdd, vars associations */
if (MaxNLDelay < MaxCurrentNLDelay)
{
    SetGnlMapNodeInfoBestCell (Node, BestCell);
    SetGnlMapNodeInfoBestArrivalR (Node, MaxNLDelayR);
    SetGnlMapNodeInfoBestArrivalF (Node, MaxNLDelayF);

    SetGnlMapNodeInfoTransitionR (Node, MaxNTransitionR);
    SetGnlMapNodeInfoTransitionF (Node, MaxNTransitionF);

    SetGnlMapNodeInfoBestBdd (Node, NewBdd);

    if (GnlSaveInstance (Node, SupportNodes))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlBestNLDelayMapNode
 */
/*
 * This is the main procedure for the NL-delay Mapping. It looks for      */
/* the best mapping starting from node 'Node' and returns the best      */
/* NL-delay seen so far. The NL-delay information is also stored in the */
/* field 'GnlMapNodeInfoBestNLDelay' of node 'Node' in order to pick it */
/* up if we recall this function on the same Node.                      */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise.        */
*/
GNL_STATUS GnlBestNLDelayMapNode (Node, Capacitance, ArrivalR, ArrivalF,
                                  TransitionR, TransitionF, Done)
{
    GNL_NODE      Node;
    float         Capacitance;
    float         *ArrivalR;
    float         *ArrivalF;
    float         *TransitionR;
    float         *TransitionF;
    int          *Done;
{
    GNL_SNODE     SupportNode;
    GNL_VAR       Var;
    GNL_MAP_NODE_INFO NewMapNodeInfo;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
```

gnlmap.c

```
{  
    Var = (GNL_VAR) GnlNodeSons (Node);  
  
    /* Either a primary input or net connected to black box */  
    if (!GnlVarFunction (Var))  
    {  
        *Done = 1;  
        *ArrivalR = 0.0;  
        *ArrivalF = 0.0;  
        *TransitionR = 0.0; /* Inputs have 0.0 transition delay */  
        *TransitionF = 0.0; /* Inputs have 0.0 transition delay */  
        return (GNL_OK);  
    }  
  
    Node = GnlFunctionOnSet (GnlVarFunction (Var));  
  
    *ArrivalR = GnlMapNodeInfoBestArrivalR (Node);  
    *ArrivalF = GnlMapNodeInfoBestArrivalF (Node);  
    *TransitionR = GnlMapNodeInfoTransitionR (Node);  
    *TransitionF = GnlMapNodeInfoTransitionF (Node);  
  
    return (GNL_OK);  
}  
  
if (GnlNodeOp (Node) == GNL_CONSTANTE)  
{  
    *Done = 1;  
    *ArrivalR = 0.0;  
    *ArrivalF = 0.0;  
    *TransitionR = 0.0;  
    *TransitionF = 0.0;  
    return (GNL_OK);  
}  
  
if (((GnlMapNodeInfoBestArrivalR (Node) != 0.0) &&  
     (GnlMapNodeInfoBestArrivalR (Node) != MAX_NL_DELAY_VALUE)) ||  
     ((GnlMapNodeInfoBestArrivalF (Node) != 0.0) &&  
     (GnlMapNodeInfoBestArrivalF (Node) != MAX_NL_DELAY_VALUE)))  
{  
    *Done = 1;  
    *ArrivalR = GnlMapNodeInfoBestArrivalR (Node);  
    *ArrivalF = GnlMapNodeInfoBestArrivalF (Node);  
    *TransitionR = GnlMapNodeInfoTransitionR (Node);  
    *TransitionF = GnlMapNodeInfoTransitionF (Node);  
    return (GNL_OK);  
}  
  
SetGnlMapNodeInfoBestArrivalR (Node, MAX_NL_DELAY_VALUE);  
SetGnlMapNodeInfoBestArrivalF (Node, MAX_NL_DELAY_VALUE);  
  
SupportNode = GnlMapNodeInfoSNodes (Node);  
  
if (GnlGetBestNLDelayCellOnNode (Node, Capacitance, SupportNode, 0,  
                                BListSize (GnlNodeSons (Node)),  
                                SupportNode, Done))
```

gnlmap.c

```
    return (GNL_MEMORY_FULL);

/* we were not able to map everything. */  
if ((*Done) == 0)  
{  
    return (GNL_OK);  
}  
  
*ArrivalR = GnlMapNodeInfoBestArrivalR (Node);  
*ArrivalF = GnlMapNodeInfoBestArrivalF (Node);  
*TransitionR = GnlMapNodeInfoTransitionR (Node);  
*TransitionF = GnlMapNodeInfoTransitionF (Node);  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGetBestDepthCellOnNode */  
/*-----*/  
/* This procedure is the main Mapping Time function. It looks for all */  
/* the cells which can realize the functionality starting from GNL_NODE */  
/* 'Node' and which leaves are the ones in 'SupportNodes'. It select */  
/* the best cell which involves the best overall Depth mapping. Recursiv*/  
/* call is performed on other functionalities starting from Node (which */  
/* are still defined by the support 'NewSupportNodes'. Indirect */  
/* recursive call is also performed on 'GnlBestDepthMapNode' when a cell*/  
/* has been found. This call is perfomed on all the Node of the current */  
/* support. The best Depth is stored in field GnlMapNodeInfoBestDepth */  
/* of 'Node'. */  
/* 'Done' is 1 if the mapper was able to map and '0' otherwise. */  
/*-----*/  
GNL_STATUS GnlGetBestDepthCellOnNode (Node, SupportNodes, Index,  
                                      SizeSupportNodes, Rec, Done)  
{  
    GNL_NODE      Node;  
    GNL_SNODE     SupportNodes;  
    int           SizeSupportNodes;  
    int           Index;  
    int           Rec;  
    int           *Done;  
    int           i;  
    GNL_VAR       VarI;  
    GNL_SNODE     SNodes;  
    int           CellFound;  
    BDD_PTR       NewBddPtr;  
    GNL_STATUS    Status;  
    BDD           NewBdd;  
    GNL_NODE      NodeI;  
    GNL_SNODE     NewSupportNodes;  
    GNL_SNODE     SNodei;  
    int           SizeNewSupport;  
    int           CDone;  
    BLIST         SortedVars;  
    float          Area;  
    float          NArea;  
    LIB_DERIVE_CELL BestCell;
```

```

int NDepth;
int Depth;
int MaxDepth;

#ifndef STAT
    G_NB_TARGET_FUNC++;
#endif

*Done = 0;

if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
{
    *Done = 1;
    return (GNL_OK);
}

SNodes = SupportNodes;
i = 0;
while (SNodes != NULL)
{
    NodeI = GnlSNodeNode (SNodes);

    /* It is not necesary to expanse Nodes with index less than      */
    /* 'Index' because it has been already done.                      */
    if ((i >= Index) &&
        (GnlNodeOp (NodeI) != GNL_VARIABLE) &&
        (GnlNodeOp (NodeI) != GNL_CONSTANTE))
    {
        SizeNewSupport = SizeSupportNodes +
            BListSize (GnlNodeSons (NodeI))-1;

        if (SizeNewSupport > G_MaxCellSizeSupport)
        {
            i++;
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }

        GnlExpanseSupportNode (SupportNodes, i, &NewSupportNodes,
                               &SNodei);

        if (GnlGetBestDepthCellOnNode (Node, NewSupportNodes, i,
                                      SizeNewSupport, Rec+1,
                                      &CDone))
            return (GNL_MEMORY_FULL);

        *Done |= CDone;

        GnlDeleteSupportNode (SupportNodes, NewSupportNodes, i,
                              SNodes,
                              SizeNewSupport-SizeSupportNodes);

        if (!G_Effort && CDone)
            break;
    }
}

```

```

        i++;
        SNodes = GnlSNodeNext (SNodes);
    }

/* We bind the Nodes of the support to bdd primary inputs of the tech*/
/* library. */
if (GnlBindSupportWithLibVars (SupportNodes))
    return (GNL_MEMORY_FULL);

if ((Status = GnlBuildDirectBddOnMapNode (Node)))
    return (Status);

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);
NewBddPtr = GetBddPtrFromBdd (NewBdd);

/* The Bdd is simply a constant. */
if ((NewBdd == bdd_one ()) || (NewBdd == bdd_zero ()))
{
    *Done = 1;
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    SetGnlMapNodeInfoBestDepth (Node, 0);
    return (GNL_OK);
}

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));

/* If no cell found we rebuild the Bdd with a pseudo canonical order */
if (!CellFound)
{
    /* We compute the signature of the Node tree from the previous      */
    /* computed Bdd. */                                                 */
    if (GnlComputeVarsSignatureFromBdd (G_LibGnl, NewBdd))
        return (GNL_MEMORY_FULL);

    if (GnlExtractSortedLibVars (SupportNodes, &SortedVars))
        return (GNL_MEMORY_FULL);

    GnlReassignLibVars (SupportNodes, SortedVars);

    if ((Status = GnlBuildDirectBddOnMapNode (Node)))
        return (Status);
}

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);
NewBddPtr = GetBddPtrFromBdd (NewBdd);

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));

```

```

        BListQuickDelete (&SortedVars);
    }

    if (CellFound)
    {
#ifdef STAT
        G_NB_CELL_FOUND++;
#endif
    }

    *Done = 1;

    /* Computing the best Depth taking this cell as solution */  

    GnlGetBestDepthCell (LibBddInfoDeriveCells (NewBddPtr),  

                         NewBdd, &BestCell, &Area, &Depth);

    MaxDepth = 0;
    SNodes = SupportNodes;
    while (SNodes != NULL)
    {
        if (GnlBestDepthMapNode (GnlSNodeNode (SNodes), &NArea,  

                               &NDepth, Rec, Done))
            return (GNL_MEMORY_FULL);

        /* Impossible to map 'GnlSNodeNode (SNodes)' */  

        if (!(*Done))
            return (GNL_OK);

        if (NDepth > MaxDepth)
            MaxDepth = NDepth;

        Area += NArea;

        /* We cope since we have already found a better solution */  

        if (MaxDepth+Depth > GnlMapNodeInfoBestDepth (Node))
            break;

        /* We cope since we have already found a better solution */  

        if ((MaxDepth+Depth == GnlMapNodeInfoBestDepth (Node)) &&  

            (Area >= GnlMapNodeInfoBestArea (Node)))
            break;

        SNodes = GnlSNodeNext (SNodes);
    }

    /* If better then save everything: Area, Bdd, vars associations */  

    if ((MaxDepth+Depth < GnlMapNodeInfoBestDepth (Node)) ||  

        ((MaxDepth+Depth == GnlMapNodeInfoBestDepth (Node)) &&  

        (Area < GnlMapNodeInfoBestArea (Node))))
    {
        SetGnlMapNodeInfoBestDepth (Node, MaxDepth+Depth);
        SetGnlMapNodeInfoBestArea (Node, Area);
        SetGnlMapNodeInfoBestBdd (Node, NewBdd);
        if (GnlSaveInstance (Node, SupportNodes))
            return (GNL_MEMORY_FULL);
    }
}

```

```

        return (GNL_OK);
    }

/*
 * GnlBestDepthMapNode
 */
/*
 * This is the main procedure for the Mapping Depth option. It looks for*
 * the best mapping starting from node 'Node' and returns the best Depth*
 * seen so far. The Depth information is also stored in the field *
 * 'GnlMapNodeInfoBestDepth' of node 'Node' in order to pick it up if   */
/* we recall this function on the same Node.                           */
/* 'Done' is 1 if the mapper was able to map and '0' otherwise.      */
/*
GNL_STATUS GnlBestDepthMapNode (Node, Area, Depth, Rec, Done)
    GNL_NODE Node;
    float     *Area;
    int       *Depth;
    int       Rec;
    int       *Done;
{
    GNL_SNODE      SupportNode;

    if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
        (GnlNodeOp (Node) == GNL_CONSTANTE))
    {
        *Done = 1;
        *Depth = 0;
        *Area = 0.0;
        return (GNL_OK);
    }

    if ((GnlMapNodeInfoBestDepth (Node) != 0) &&
        (GnlMapNodeInfoBestDepth (Node) != MAX_DEPTH_VALUE))
    {
        *Done = 1;
        *Depth = GnlMapNodeInfoBestDepth (Node);
        *Area = GnlMapNodeInfoBestArea (Node);
        return (GNL_OK);
    }

    SetGnlMapNodeInfoBestDepth (Node, MAX_DEPTH_VALUE);
    SetGnlMapNodeInfoBestArea (Node, MAX_AREA_VALUE);

    SupportNode = GnlMapNodeInfoSNodes (Node);

    if (GnlGetBestDepthCellOnNode (Node, SupportNode, 0,
                                  BListSize (GnlNodeSons (Node)), Rec,
                                  Done))
        return (GNL_MEMORY_FULL);

    /* we were not able to map everything.                                */
    if ((*Done) == 0)
    {

```

gnlmap.c

```
        return (GNL_OK);
    }

*Depth = GnlMapNodeInfoBestDepth (Node);
*Area = GnlMapNodeInfoBestArea (Node);

    return (GNL_OK);
}

/*-----
/* GnlGetBestNetCellOnNode
/*-----
/* This procedure is the main Mapping Net function. It looks for all
/* the cells which can realize the functionality starting from GNL_NODE
/* 'Node' and which leaves are the ones in 'SupportNodes'. It select
/* the best cell which involves the best overall Net mapping. Recursive
/* call is performed on other functionalities starting from Node (which
/* are still defined by the support 'NewSupportNodes'. Indirect
/* recursive call is also performed on 'GnlBestDepthMapNode' when a cell
/* has been found. This call is performed on all the Node of the current
/* support. The best Net is stored in field GnlMapNodeInfoBestNet
/* of 'Node'.
/* 'Done' is 1 if the mapper was able to map and '0' otherwise.
/*-----*/
GNL_STATUS GnlGetBestNetCellOnNode (Node, SupportNodes, Index,
                                    SizeSupportNodes, Rec, Done)
{
    GNL_NODE      Node;
    GNL_SNODE     SupportNodes;
    int           SizeSupportNodes;
    int           Index;
    int           Rec;
    int           *Done;

    int           i;
    GNL_VAR       VarI;
    GNL_SNODE     SNodes;
    int           CellFound;
    BDD_PTR       NewBddPtr;
    GNL_STATUS    Status;
    BDD           NewBdd;
    GNL_NODE      NodeI;
    GNL_SNODE     NewSupportNodes;
    GNL_SNODE     SNodei;
    int           SizeNewSupport;
    int           CDone;
    BLIST          SortedVars;
    int           Depth;
    int           NDepth;
    LIB_DERIVE_CELL BestCell;
    int           NNbNet;
    int           NbNet;
    int           MaxDepth;

#endif STAT
    G_NB_TARGET_FUNC++;
#endif
```

```

*Done = 0;

if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
{
    *Done = 1;
    return (GNL_OK);
}
SNodes = SupportNodes;
i = 0;
while (SNodes != NULL)
{
    NodeI = GnlSNodeNode (SNodes);

    /* It is not necesary to expanse Nodes with index less than */
    /* 'Index' because it has been already done. */
    if ((i >= Index) &&
        (GnlNodeOp (NodeI) != GNL_VARIABLE) &&
        (GnlNodeOp (NodeI) != GNL_CONSTANTE))
    {
        SizeNewSupport = SizeSupportNodes +
                        BListSize (GnlNodeSons (NodeI))-1;

        if (SizeNewSupport > G_MaxCellSizeSupport)
        {
            i++;
            SNodes = GnlSNodeNext (SNodes);
            continue;
        }

        GnlExpanseSupportNode (SupportNodes, i, &NewSupportNodes,
                               &SNodei);

        if (GnlGetBestNetCellOnNode (Node, NewSupportNodes, i,
                                    SizeNewSupport, Rec+1,
                                    &CDone))
            return (GNL_MEMORY_FULL);

        *Done |= CDone;

        GnlDeleteSupportNode (SupportNodes, NewSupportNodes, i,
                              SNodes,
                              SizeNewSupport-SizeSupportNodes);

        if (!G_Effort && CDone)
            break;
    }

    i++;
    SNodes = GnlSNodeNext (SNodes);
}

/* We bind the Nodes of the support to bdd primary inputs of the tech*/
/* library. */
if (GnlBindSupportWithLibVars (SupportNodes))
    return (GNL_MEMORY_FULL);

```

```

if ((Status = GnlBuildDirectBddOnMapNode (Node)))
    return (Status);

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);
NewBddPtr = GetBddPtrFromBdd (NewBdd);

/* The Bdd is simply a constant. */
if ((NewBdd == bdd_one ()) || (NewBdd == bdd_zero ()))
{
    *Done = 1;
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    SetGnlMapNodeInfoBestNet (Node, 0);
    SetGnlMapNodeInfoBestDepth (Node, 0);
    return (GNL_OK);
}

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));

/* If no cell found we rebuild the Bdd with a pseudo canonical order */
if (!CellFound)
{
    /* We compute the signature of the Node tree from the previous      */
    /* computed Bdd.                                                       */
    if (GnlComputeVarsSignatureFromBdd (G_LibGnl, NewBdd))
        return (GNL_MEMORY_FULL);

    if (GnlExtractSortedLibVars (SupportNodes, &SortedVars))
        return (GNL_MEMORY_FULL);

    GnlReassignLibVars (SupportNodes, SortedVars);

    if ((Status = GnlBuildDirectBddOnMapNode (Node)))
        return (Status);
}

#ifndef STAT
    G_ROBDD_BUILT++;
#endif

NewBdd = GnlMapNodeInfoBdd (Node);
NewBddPtr = GetBddPtrFromBdd (NewBdd);

CellFound = (GetBddPtrHook (NewBddPtr) &&
             BListSize (LibBddInfoDeriveCells (NewBddPtr)));

BListQuickDelete (&SortedVars);
}

if (CellFound)
{
#ifndef STAT
    G_NB_CELL_FOUND++;
}

```

gnlmap.c

```
#endif

*Done = 1;

/* Computing the best Net taking this cell as solution */  
GnlGetBestNetCell (LibBddInfoDeriveCells (NewBddPtr),  
                    NewBdd, &BestCell, &Depth, &NbNet);

MaxDepth = 0;
SNodes = SupportNodes;
while (SNodes != NULL)
{
    if (GnlBestNetMapNode (GnlSNodeNode (SNodes), &NDepth,
                           &NNbNet, Rec, Done))
        return (GNL_MEMORY_FULL);

    /* Impossible to map 'GnlSNodeNode (SNodes)' */  
    if (!(*Done))
        return (GNL_OK);

    NbNet += NNbNet;

    if (NDepth > MaxDepth)
        MaxDepth = NDepth;

    /* We cope since we have already found a better solution */  
    if (NbNet > GnlMapNodeInfoBestNet (Node))
        break;

    SNodes = GnlSNodeNext (SNodes);
}

/* If better then save everything: Area, Bdd, vars associations */  
if ((NbNet < GnlMapNodeInfoBestNet (Node)) ||
    (NbNet == GnlMapNodeInfoBestNet (Node)) &&
    (MaxDepth+Depth < GnlMapNodeInfoBestDepth (Node)))
{
    SetGnlMapNodeInfoBestNet (Node, NbNet);
    SetGnlMapNodeInfoBestDepth (Node, MaxDepth+Depth);
    SetGnlMapNodeInfoBestBdd (Node, NewBdd);
    if (GnlSaveInstance (Node, SupportNodes))
        return (GNL_MEMORY_FULL);
}
}

return (GNL_OK);
}

/*-----*/  
/* GnlBestNetMapNode */  
/*-----*/  
/* This is the main procedure for the Mapping Nets option. It looks for */  
/* the best mapping starting from node 'Node' and returns the best Nets */  
/* seen so far. The Net information is also stored in the field */  
/* 'GnlMapNodeInfoBestNet' of node 'Node' in order to pick it up if */  
/* we recall this function on the same Node. */  
/*-----*/
```

gnlmap.c

```
/* 'Done' is 1 if the mapper was able to map and '0' otherwise. */  
/*-----*/  
GNL_STATUS GnlBestNetMapNode (Node, Depth, NbNet, Rec, Done)  
{  
    GNL_NODE      Node;  
    int          *Depth;  
    int          *NbNet;  
    int          Rec;  
    int          *Done;  
  
    GNL_SNODE     SupportNode;  
  
    if ((GnlNodeOp (Node) == GNL_VARIABLE) ||  
        (GnlNodeOp (Node) == GNL_CONSTANTE))  
    {  
        *Done = 1;  
        *Depth = 0;  
        *NbNet = 0;  
        return (GNL_OK);  
    }  
  
    if ((GnlMapNodeInfoBestNet (Node) != 0) &&  
        (GnlMapNodeInfoBestNet (Node) != MAX_NET_VALUE))  
    {  
        *Done = 1;  
        *Depth = GnlMapNodeInfoBestDepth (Node);  
        *NbNet = GnlMapNodeInfoBestNet (Node);  
        return (GNL_OK);  
    }  
  
    SetGnlMapNodeInfoBestNet (Node, MAX_NET_VALUE);  
    SetGnlMapNodeInfoBestDepth (Node, MAX_DEPTH_VALUE);  
  
    SupportNode = GnlMapNodeInfoSNodes (Node);  
  
    if (GnlGetBestNetCellOnNode (Node, SupportNode, 0,  
                                BListSize (GnlNodeSons (Node)), Rec,  
                                Done))  
        return (GNL_MEMORY_FULL);  
  
    /* we were not able to map everything. */  
    if ((*Done) == 0)  
    {  
        return (GNL_OK);  
    }  
  
    *NbNet = GnlMapNodeInfoBestNet (Node);  
    *Depth = GnlMapNodeInfoBestDepth (Node);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGetMinCapacitance */  
/*-----*/  
/* Returns the minimum capacitance between all the input/inout pins of */  
/* the cell 'MotherCell'. */  
/*-----*/
```

gnlmap.c

```
/*-----*/
float GnlGetMinCapacitance (MotherCell)
    LIB_CELL MotherCell;
{
    float      MinCapa;
    int         i;
    LIBC_PIN Pins;
    LIBC_CELL   InvCell;

    MinCapa = 1000000.0;
    InvCell = LibHCellLibcCell (MotherCell);
    Pins = LibCellPins (InvCell);

    for (;Pins != NULL; Pins = LibPinNext(Pins))
    {
        if ((LibPinDirection (Pins) == INPUT_E) ||
            (LibPinDirection (Pins) == INOUT_E))
        {
            if (LibPinCapa (Pins) < MinCapa)
                MinCapa = LibPinCapa (Pins);
        }
    }

    if (MinCapa == 1000000.0)
        return (0.0);

    return (MinCapa);
}

/*-----*/
/* GnlMapGnl */
/*-----*/
/* Main mapping procedure which maps a Gnl according to a library of */
/* cells 'GnlLib'. Done is 1 if everything was mapped and 0 otherwise. */
/* 'GnlLib' is the library of cells that we map on. The criterion under */
/* which the mapping is performed is given by the global variable: */
/* 'GnlEnvCriterion ()'. */
/*-----*/
GNL_STATUS GnlMapGnl (Nw, Gnl, GnlLibc, Effort, Done)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    LIBC_LIB GnlLibc;
    int              Effort;           /* 1 -> HIGH, 0 -> LOW/MEDIUM      */
    int              *Done;
{
    int                  i;
    GNL_VAR             VarI;
    GNL_FUNCTION        FunctionI;
    GNL_NODE            Node;
    float               FArea;
    float               FArrivalR;
    float               FArrivalF;
    float               FTransitionR;
    float               FTransitionF;
```

gnlmap.c

```
int FDepth;
int MaxLibInputs;
float Capacitance;
GNL_LIB GnlLib;
LIBC_WIRE_LOAD_SELECT WireLoadSelect;
int Fanout;
float Length;

G_Gnl = Gnl;
G_GnlLibc = GnlLibc;
GnlLib = (GNL_LIB)LibHook (GnlLibc);
G_GnlLib = GnlLib;
G_LibGnl = GnlHLibGnl (GnlLib);
G_LibInputs = GnlInputs (G_LibGnl);

/* By default the maximum fanin size of considered cells is          */
/* GNL_DEFAULT_FANIN_CELL.                                              */
MaxLibInputs = GNL_DEFAULT_FANIN_CELL;

G_MaxCellSizeSupport = (BListSize (G_LibInputs) > MaxLibInputs ?
                        MaxLibInputs :
                        BListSize (G_LibInputs));
if (GnlEnvMaxCellSizeSupport ())
    G_MaxCellSizeSupport = GnlEnvMaxCellSizeSupport ();

/* we intialize the library util variables with the default operating*/
/* conditions.                                                       */
LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, NULL);

switch (GnlEnvCriterion ()) {
    case GNL_AREA:
        G_BestInverter = GetBestAreaLibInverter (GnlLib);
        fprintf (stderr, "      o Criterion = AREA\n");
        break;

    case GNL_TIMING:
        /* Call later the best inverter selection on timing. */
        G_BestInverter = GetBestAreaLibInverter (GnlLib);
        fprintf (stderr, "      o Criterion = TIMING\n");
        break;

    case GNL_NB_NETS:
        /* Call later the best inverter selection on timing. */
        G_BestInverter = GetBestAreaLibInverter (GnlLib);
        fprintf (stderr, "      o Criterion = NB. NETS\n");
        break;

    case GNL_NL_DELAY:
        GnlSortFunctionsOnInvertLevel (Gnl);

        WireLoadSelect = LibGetWireLoadSelectFromName (GnlLibc,
                                                       (char *)NULL);
        G_WireLoad = LibGetWireLoadFromAreaAndWireLS (WireLoadSelect,
                                                      GnlNetworkNLDelayArea (Nw), GnlLibc);
```

gnlmap.c

```
Fanout = 1;
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
G_WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad,
                                              Length, GnlLibc);

/* we force the output capacitance to be 0.0          */
Capacitance = 0.0 + G_WireCapa;

/* Call later the best inverter selection on timing. */
G_BestInverter = GetBestAreaLibInverter (GnlLib);
fprintf (stderr, "      o Criterion = NL DELAY\n");
break;

case GNL_POWER:
    G_BestInverter = GetBestPowerLibInverter (GnlLib);

    /* Extracting an initial capacitance for the current node */
    /* we take the capacitance of a single inverter.           */
    Capacitance = GnlGetMinCapacitance (G_BestInverter);
    fprintf (stderr, "      o Criterion = POWER\n");
    break;
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    Node = GnlFunctionOnSet (FunctionI);

    GnlResetGnlNodeHook (Node);

    if (!GnlEnvLog ())
    {
        fprintf (stderr, "%c      o Mapping Equation [%d/%d]", 13, i,
                 BListSize (GnlFunctions (Gnl)));
        fflush (stderr);
    }

/* we add the mandatory structure in order to store all infos */
/* allowing to do the mapping.                                */
if (GnlAddMapNodeInfo (Node))
    return (GNL_MEMORY_FULL);

/* Either Area mapping, Timing mapping or Power Mapping.     */
switch (GnlEnvCriterion ()) {
    case GNL_AREA:
        if (GnlBestAreaMapNode (Node, &FArea, &FDepth, Done))
            return (GNL_MEMORY_FULL);
        break;

    case GNL_TIMING:
        if (GnlBestDepthMapNode (Node, &FArea, &FDepth, 0,
                               Done))
            return (GNL_MEMORY_FULL);
        break;
}
```

gnlmap.c

```
case GNL_NB_NETS:
    if (GnlBestNetMapNode (Node, &FDepth, &FDepth, 0,
                           Done))
        return (GNL_MEMORY_FULL);
    break;

case GNL_POWER:
    if (GnlBestPowerMapNode (Node, Capacitance,
                             &FArea,
                             &FDepth, 0, Done))
        return (GNL_MEMORY_FULL);
    break;

case GNL_NL_DELAY:
    /* Actually the capacitance has been already           */
    /* pre-computed.                                         */
    if (GnlVarOutCapa (VarI) != 0.0)
        Capacitance = GnlVarOutCapa (VarI);

    if (GnlBestNLDelayMapNode (Node, Capacitance,
                               &FArrivalR, &FArrivalF,
                               &FTransitionR, &FTransitionF,
                               Done))
        return (GNL_MEMORY_FULL);
    break;

}

/* Free SNode field of each Gnl node in the tree starting from      */
/* 'Node'.                                                       */
/* This helps to gain some relative memory                         */
GnlFreeSNodeOfMapNodeInfo (Node);

if (!(*Done))
{
    return (GNL_OK);
}
}

if (!GnlEnvLog () && BListSize (GnlFunctions (Gnl)))
{
    fprintf (stderr, "%c      o Mapping Equation [%d/%d]\n",
            13, i, BlistSize (GnlFunctions (Gnl)));
    fflush (stderr);
}

return (GNL_OK);
}

/*
 * GnlGetOriginalVar
 */
/*
 * This function returns the ORIGINAL GNL_VAR of the cell 'MotherCell'  */
/* which corresponds to one of the original port.                      */
/* Example:                                                               */

```

gnlmap.c

```
/*
 *      LIB_DERIVE_CELL    : cell        -> $i_2 $i_3 $i     $i_1          */
 *      LIB_CELL           : MotherCell   -> $i     $i_1 $i_2 $i_3          */
 *      cell (library)          A       B       C       D          */
 */
/* If 'Var' = '$i_3' then the return variable is 'D'.          */
/*-----*/
GNL_VAR GnlGetOriginalVar (Cell, MotherCell, Var)
    LIB_DERIVE_CELL    Cell;
    LIB_CELL           MotherCell;
    GNL_VAR            Var;
{
    int                 i;
    int                 Index;
    GNL_VAR             ResVar;

    Index = -1;
    for (i=0; i<BListSize (LibDeriveCellInputs (Cell)); i++)
    {
        if (Var == (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i))
        {
            Index = i;
            break;
        }
    }

    if (Index == -1)
        return (NULL);

    ResVar = (GNL_VAR)BListElt (LibHCellInputsOrigin (MotherCell), Index);
    return (ResVar);
}

/*
 *-----*
 * GnlCreateInverterGateFromNode
 *-----*/
GNL_STATUS GnlCreateInverterGateFromNode (Gnl, BestInv, OutVar, Node)
    GNL             Gnl;
    LIB_CELL        BestInv;
    GNL_VAR         OutVar;
    GNL_NODE        Node;
{
    GNL_VAR         Var;
    GNL_VAR         NewVar;
    GNL_FUNCTION    NewFunction;
    BDD             Bdd;
    BDD_PTR         BddPtr;
    BLIST           ListDeriveCells;
    GNL_NODE        NewNode;
    GNL_NODE        NodeNot;
    GNL_MAP_NODE_INFO NewMapNodeInfo;
    LIB_DERIVE_CELL FirstCell;
    LIB_DERIVE_CELL BestCell;
    BLIST           newList;
    float           Area;
    int              Depth;
```

gnlmap.c

```
int i;

if ((GnlNodeOp (Node) == GNL_VARIABLE) ||
    (GnlNodeOp (Node) == GNL_CONSTANTE))
    return (GNL_OK);

Var = GnlMapNodeInfoCutVar (Node);
if (Var != OutVar)
    return (GNL_OK);

Bdd = GnlMapNodeInfoBestBdd (Node);

if (!Bdd)
    return (GNL_OK);

BddPtr = GetBddPtrFromBdd (Bdd);
if ((Bdd == bdd_one ()) ||
    (Bdd == bdd_zero ()))
    return (GNL_OK);

ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
GnlGetBestAreaCell (LibBddInfoDeriveCells (BddPtr), Bdd, &BestCell,
                     &Area, &Depth);

/* There is a complemented gate so we create a new var to store this */
/* complement. */
if (LibDeriveCellBdd (BestCell) != Bdd)
{
    if (GnlCreateNode (Gnl, NULL, &NewNode))
        return (GNL_MEMORY_FULL);

    /* NewNode is a copy of original node 'Node'. */
    SetGnlNodeOp (NewNode, GnlNodeOp (Node));
    if (BListCreateWithSize (BListSize (GnlNodeSons (Node)), &NewList))
        return (GNL_MEMORY_FULL);
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        if (BlistAddElt (NewList, BListElt (GnlNodeSons (Node), i)))
            return (GNL_MEMORY_FULL);
    }
    SetGnlNodeSons (NewNode, NewList);
    SetGnlNodeLineNumber (NewNode, GnlNodeLineNumber (Node));

    if (GnlCreateNodeNot (Gnl, NewNode, &NodeNot))
        return (GNL_MEMORY_FULL);
    SetGnlNodeHook (NodeNot, GnlNodeHook (Node));

    /* The complement node has then the same Bdd as the Best cell */
    SetGnlMapNodeInfoBestBdd (NodeNot, LibDeriveCellBdd (BestCell));
    SetGnlMapNodeInfoBestArea (NodeNot, GnlMapNodeInfoBestArea (Node) -
                               LibHCellArea (BestInv));

    if (GnlCreateUniqueVar (Gnl, "IM", &NewVar))
        return (GNL_MEMORY_FULL);
    SetGnlVarType (NewVar, GNL_VAR_ORIGINAL);
```

```

SetGnlMapNodeInfoCutVar (NodeNot, NewVar);

if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (NewVar, NewFunction);
SetGnlFunctionOnSet (NewFunction, NodeNot);

if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);

SetGnlNodeOp (Node, GNL_NOT);
BSize (GnlNodeSons (Node)) = 0;
if (BListAddElt (GnlNodeSons (Node), (int)NodeNot))
    return (GNL_MEMORY_FULL);

if (GnlMapNodeInfoCreate (&NewMapNodeInfo))
    return (GNL_MEMORY_FULL);
SetGnlNodeHook (Node, NewMapNodeInfo);
ListDeriveCells = LibHCellDeriveCells (BestInv);
FirstCell = (LIB_DERIVE_CELL)BListElt (ListDeriveCells, 0);
SetGnlMapNodeInfoBdd (Node, LibDeriveCellBdd (FirstCell));
SetGnlMapNodeInfoBestBdd (Node, LibDeriveCellBdd (FirstCell));
SetGnlMapNodeInfoCutVar (Node, OutVar);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlMapNodeInfoLibVarSupport (Node, NewList);

/* an inverter so we pick up the first lib var. */
if (BListAddElt (NewList, BListElt (G_LibInputs, 0)))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlMapNodeInfoNodeSupport (Node, NewList);
if (BListAddElt (NewList, BListElt (GnlNodeSons (Node), 0)))
    return (GNL_MEMORY_FULL);
SetGnlMapNodeInfoBestArea (Node, LibHCellArea (BestInv));

}

return (GNL_OK);
}

/*
 * GnlCreateInverterGate
 */
GNL_STATUS GnlCreateInverterGate (Gnl, GnlLib)
    GNL           Gnl;
    GNL_LIB      GnlLib;

```

gnlmap.c

```
{  
    int i;  
    GNL_VAR VarI;  
    GNL_FUNCTION FunctionI;  
    LIB_CELL BestInv;  
    int NbFunctions;  
  
    switch (GnlEnvCriterion ()) {  
        case GNL_AREA:  
            BestInv = GetBestAreaLibInverter (GnlLib);  
            break;  
  
        case GNL_TIMING:  
            /* Call later the best inverter selection on timing. */  
            BestInv = GetBestAreaLibInverter (GnlLib);  
            break;  
  
        case GNL_NB_NETS:  
            BestInv = GetBestAreaLibInverter (GnlLib);  
            break;  
  
        case GNL_POWER:  
            BestInv = GetBestPowerLibInverter (GnlLib);  
            break;  
    }  
  
    NbFunctions = BListSize (GnlFunctions (Gnl));  
  
    for (i=0; i<NbFunctions; i++)  
    {  
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  
        FunctionI = GnlVarFunction (VarI);  
  
        if (GnlCreateInverterGateFromNode (Gnl, BestInv, VarI,  
                                           GnlFunctionOnSet (FunctionI)))  
            return (GNL_MEMORY_FULL);  
    }  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlEqualMapNode */  
/*-----*/  
/* This function returns 1 if the Node expressions are the same */  
/*-----*/  
int GnlEqualMapNode (Node1, Node2, Depth)  
{  
    GNL_NODE Node1;  
    GNL_NODE Node2;  
    int i;  
    GNL_NODE Son1;  
    GNL_NODE Son2;
```

gnlmap.c

```
if (GnlNodeOp (Node1) != GnlNodeOp (Node2))
    return (0);

if (GnlMapNodeInfoCutVar (Node1))
{
    if (GnlMapNodeInfoCutVar (Node1) == GnlMapNodeInfoCutVar (Node2))
        return (1);
    if (Depth)
        return (0);
}

if ((GnlNodeOp (Node1) == GNL_VARIABLE) ||
    (GnlNodeOp (Node1) == GNL_CONSTANTE))
{
    if (GnlNodeSons (Node1) == GnlNodeSons (Node2))
        return (1);
    else
        return (0);
}

if (Depth)
    return (0);

for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)
{
    Son1 = (GNL_NODE)BListElt (GnlNodeSons (Node1), i);
    Son2 = (GNL_NODE)BListElt (GnlNodeSons (Node2), i);
    if (!GnlEqualMapNode (Son1, Son2, Depth+1))
        return (0);
}

return (1);
}

/*-----*/
/* GnlSubstituteVarByVarInNode */
/*-----*/
void GnlSubstituteVarByVarInNode (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE Son;
    GNL_VAR Var;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        if (GnlVarHook (Var))
            SetGnlNodeSons (Node, (BLIST)GnlVarHook (Var));
        return ;
    }

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return ;

    /* If the node corresponds to a mapping cut var. */
}
```

gnlmap.c

```
if (GnlMapNodeInfoCutVar (Node))
{
    /* We replace one function by another one. We stop at this level */
    /* the recursion.                                                 */
    if (GnlVarHook (GnlMapNodeInfoCutVar (Node)))
    {
        SetGnlMapNodeInfoCutVar (Node,
            GnlVarHook (GnlMapNodeInfoCutVar (Node)));
        return;
    }
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    Son = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    GnlSubstituteVarByVarInNode (Son);
}

/*
 *-----*
 * GnlSubstituteVarByVar                         */
/*-----*/
/* This function substitutes the Variable 'Var1' by the Variable 'Var2' */
/* everywhere it occurs.                                         */
/*-----*/
void GnlSubstituteVarByVar (Gnl, Var1, Var2)
    GNL           Gnl;
    GNL_VAR      Var1;
    GNL_VAR      Var2;
{
    int          i;
    GNL_VAR      VarI;
    GNL_FUNCTION FunctionI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        GnlSubstituteVarByVarInNode (GnlFunctionOnSet (FunctionI));
    }
}

/*
 *-----*
 * GnlFactorizeGates                           */
/*-----*/
/* This procedure is a post-mapping optimization which take a look at   */
/* the Map variables which have the same right hand side expression and */
/* factorize them.                                                 */
/*-----*/
GNL_STATUS GnlFactorizeGates (Gnl, NbOriginalFunctions)
    GNL           Gnl;
    int          NbOriginalFunctions;
{
    int          i;
```

gnlmap.c

```
int          j;
GNL_VAR      VarI;
GNL_VAR      VarJ;
GNL_FUNCTION FunctionI;
GNL_FUNCTION FunctionJ;
BLIST       HashTableNames;
BLIST       BucketI;
int         MaxBound;

if (BListSize (GnlFunctions (Gnl)) > MAX_FACTORIZE_FUNC)
    return (GNL_OK);

GnlResetVarHook (Gnl);

for (i=NbOriginalFunctions; i<BListSize (GnlFunctions (Gnl)); i++)
{
    if (!GnlEnvLog ())
    {
        fprintf (stderr, "%c Post mapping optimization [%d/%d]", 13, i,
                 BListSize (GnlFunctions (Gnl)));
        fflush (stderr);
    }

    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    if (VarI != GnlMapNodeInfoCutVar (GnlFunctionOnSet (FunctionI)))
        continue;

    MaxBound = (BListSize (GnlFunctions (Gnl)) > i+2000 ? i+2000 :
                BListSize (GnlFunctions (Gnl)));
    for (j=i+1; j<MaxBound; j++)
    {
        VarJ = (GNL_VAR)BListElt (GnlFunctions (Gnl), j);

        /* 'VarJ' has been already replaced. */
        if (GnlVarHook (VarJ))
            continue;

        FunctionJ = GnlVarFunction (VarJ);
        if (GnlEqualMapNode (GnlFunctionOnSet (FunctionI),
                            GnlFunctionOnSet (FunctionJ), 0))
        {
            SetGnlVarHook (VarJ, VarI);
            GnlFunctionFree (GnlVarFunction (VarJ));
            SetGnlVarFunction (VarJ, NULL);
            BListDelInsert (GnlFunctions (Gnl), j+1);
            j--;
            MaxBound = (BListSize (GnlFunctions (Gnl)) >
                        i+2000 ? i+2000 :
                        BListSize (GnlFunctions (Gnl)));
        }
    }
}

if (!GnlEnvLog () &&
    (NbOriginalFunctions < BListSize (GnlFunctions (Gnl))))
```

gnlmap.c

```
{  
    fprintf (stderr, "%c Post mapping optimization [%d/%d]", 13, i,  
            BListSize (GnlFunctions (Gnl)));  
    fflush (stderr);  
    fprintf (stderr, "\n");  
}  
  
GnlSubstituteVarByVar (Gnl);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlCreateCutVarsOnNode */  
/*-----*/  
/* Creates physically a new GNL_VAR which is attached to a corresponding */  
/* GNL_NODE. This var is stored in the field of GNL_NODE node */  
/* 'GnlMapNodeInfoCutVar'. If 'RootVar' is not NULL then we do not */  
/* create a new GNL_VAR and use the one given (i.e 'RootVar'). */  
/*-----*/  
GNL_STATUS GnlCreateCutVarsOnNode (Gnl, Node, RootVar)  
{  
    GNL          Gnl;  
    GNL_NODE     Node;  
    GNL_VAR      RootVar;  
  
    int          i;  
    GNL_NODE    NodeI;  
    GNL_VAR     NewVar;  
    GNL_FUNCTION NewFunction;  
    BLIST       Sons;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
        return (GNL_OK);  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
        return (GNL_OK);  
  
    if (RootVar)  
    {  
        SetGnlMapNodeInfoCutVar (Node, RootVar);  
    }  
    else  
    {  
        if (GnlCreateUniqueVar (Gnl, "M", &NewVar))  
            return (GNL_MEMORY_FULL);  
        SetGnlVarType (NewVar, GNL_VAR_ORIGINAL);  
        SetGnlMapNodeInfoCutVar (Node, NewVar);  
  
        if (BListAddElt (GnlLocals(Gnl), (int)NewVar))  
            return (GNL_MEMORY_FULL);  
        SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);  
  
        if (GnlFunctionCreate (Gnl, NewVar, Node, &NewFunction))  
            return (GNL_MEMORY_FULL);  
    }
```

gnlmap.c

```
SetGnlVarFunction (NewVar, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);

}

if (GnlMapNodeInfoNodeSupport (Node) == NULL)
    return (GNL_OK);

Sons = GnlMapNodeInfoNodeSupport (Node);

for (i=0; i<BListSize (Sons); i++)
{
    NodeI = (GNL_NODE)BListElt (Sons, i);
    if (GnlCreateCutVarsOnNode (Gnl, NodeI, NULL))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 *-----*
 * GnlCreateCutVars
 *-----*
 * This procedure creates cut variables which correspond to the Nodes
 * which are linked to a technology cell. These cut variables are then
 * bind to their respective Node thru the field 'nlMapNodeInfoCutVar'
 *-----*
GNL_STATUS GnlCreateCutVars (Gnl)
    GNL      Gnl;
{
    int          i;
    int          NbFunctions;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;

NbFunctions = BListSize (GnlFunctions (Gnl));

for (i=0; i<NbFunctions; i++)
{
    if (!GnlEnvLog ())
    {
        fprintf (stderr, "%c Building cell [%d/%d]", 13, i, NbFunctions);
        fflush (stderr);
    }
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (GnlCreateCutVarsOnNode (Gnl, GnlFunctionOnSet (FunctionI), VarI))
        return (GNL_MEMORY_FULL);
}

if (!GnlEnvLog () && NbFunctions)
{
```

gnlmap.c

```
.     fprintf (stderr, "%c Building cell [%d/%d]", 13, i, NbFunctions);
     fflush (stderr);
     fprintf (stderr, "\n");
}

return (GNL_OK);
}

/*-----
/* GnlReconstructMappedCell                               */
/*-----*/
/* This procedure creates a GNL_USER_COMPONENT from a mapped sub-tree. */
/*-----*/
GNL_STATUS GnlReconstructMappedCell (Gnl, OutVar, Cell, ListDeriveCells,
                                     Node)
    GNL                      Gnl;
    GNL_VAR                  OutVar;
    LIB_DERIVE_CELL          Cell;
    BLIST                     ListDeriveCells;
    GNL_NODE                 Node;
{
    int                      i;
    int                      j;
    LIB_CELL                 MotherCell;
    BLIST                     Interface;
    GNL_ASSOC                NewAssoc;
    char                     *InstanceName;
    BLIST                     NodeSupport;
    BLIST                     LibVarSupport;
    GNL_VAR                  VarI;
    GNL_VAR                  VarCellII;
    GNL_VAR                  CellPort;
    GNL_USER_COMPONENT        NewUserCompo;
    GNL_NODE                 NodeJ;
    BLIST                     MapNodeInfoSupport;

MotherCell = LibDeriveCellMotherCell (Cell);

if (BListCreateWithSize (1, &Interface))
    return (GNL_MEMORY_FULL);

SetGnlInstanceId (Gnl, GnlInstanceId (Gnl)+1);
if (GnlStrAppendIntCopy ("m", GnlInstanceId (Gnl), &InstanceName))
    return (GNL_MEMORY_FULL);

if (GnlCreateUserComponent (LibHCellName (MotherCell),
                           InstanceName, NULL, Interface,
                           &NewUserCompo))
    return (GNL_MEMORY_FULL);

/* we link the component with its corresponding LIBC cell           */
SetGnlUserComponentCellDef (NewUserCompo,
                            LibHCellLibcCell (MotherCell));
SetGnlUserComponentEquivCells (NewUserCompo, ListDeriveCells);
```

```

if (GnlCreateAssoc (&NewAssoc))
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort (NewAssoc, LibHCellOutput (MotherCell));
SetGnlAssocActualPort (NewAssoc, OutVar);
if (BListAddElt (Interface, (int)NewAssoc))
    return (GNL_MEMORY_FULL);

NodeSupport = GnlMapNodeInfoNodeSupport (Node);
LibVarSupport = GnlMapNodeInfoLibVarSupport (Node);

#ifndef TRACE_BIND_CELL
fprintf (stderr, "MOTHER CELL = %s\n", LibHCellName (MotherCell));
GnlPrintNode (stderr, LibHCellFunction (MotherCell));
fprintf (stderr, "\n");
PrintListVar (LibHCellInputsOrigin (MotherCell));
fprintf (stderr, "\n");
fprintf (stderr, "DERIVE CELL\n");
PrintListVar (LibDeriveCellInputs (Cell));
bdd_print (LibDeriveCellBdd (Cell));
fprintf (stderr, "\n");
PrintListVar (GnlMapNodeInfoLibVarSupport (Node));
#endif

/* For each input of the Cell ... */
for (i=0; i<BListSize (LibDeriveCellInputs (Cell)); i++)
{
    /* We extract 'VarI' which a library variable $i<index> from the*/
    /* derive cell which matches exactly the bdd of the Node. */
    VarI = (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i);

    /* we look for the index of VarI = $i<index> e.g the value which*/
    /* is 'index' in '$i<index>'. */
    for (j=0; j<BListSize (GnlMapNodeInfoLibVarSupport (Node)); j++)
    {
        VarCellI = (GNL_VAR)
            BListElt (GnlMapNodeInfoLibVarSupport (Node), j);
        if (VarI == VarCellI)
            break;
    }

    /* We extract the node of the Gnl which is linked to the lib      */
    /* '$i<index>'. It is the element in 'GnlMapNodeInfoNodeSupport' */
    /* at location 'index'. */
    MapNodeInfoSupport = GnlMapNodeInfoNodeSupport (Node);
    NodeJ = (GNL_NODE)BListElt (MapNodeInfoSupport, j);
    if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
        VarI = (GNL_VAR)GnlNodeSons (NodeJ);
    else
        VarI = GnlMapNodeInfoCutVar (NodeJ);

    /* 'VarCellI' represents '$i<index>'. */
    CellPort = GnlGetOriginalVar (Cell, MotherCell, VarCellI);

    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, GnlVarName (CellPort));
}

```

```

        SetGnlAssocActualPort (NewAssoc, VarI);

        if (BListAddElt (Interface, (int)NewAssoc))
            return (GNL_MEMORY_FULL);
    }

/* Adding the new created component in the list of components of      */
/* original 'Gnl'.                                                 */
if (BListAddElt (GnlComponents (Gnl), (int)NewUserCompo))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----*/
/* GnlReconstructMappedFunction                                     */
/*-----*/
/* This procedure analyzes a mapped function and returns a new GNL_NODE */
/* iff this is a simple assignment or creates a new user component from */
/* the mapped function and adds it in the list of components of 'Gnl'.   */
/* For a simple assignment the new GNL_NODE is created for a new GNL   */
/* object called 'GnlTemp'. Then 'NewNode' has the value of this new   */
/* if it is a simple assignment and NULL if the mapped node inferred a  */
/* user component.                                                 */
/*-----*/
GNL_STATUS GnlReconstructMappedFunction (Gnl, TempGnl, BestInv,
                                         OutVar, Node, NewNode)
{
    GNL          Gnl;
    GNL          TempGnl;
    LIB_CELL     BestInv;
    GNL_VAR      OutVar;
    GNL_NODE     Node;
    GNL_NODE *NewNode;

    GNL_VAR      Var;
    BDD          Bdd;
    BDD_PTR      BddPtr;
    BLIST         ListDeriveCells;
    LIB_DERIVE_CELL BestCell;
    float         Area;
    int          Depth;
    GNL_USER_COMPONENT UserCompo;

    switch (GnlNodeOp (Node)) {
        case GNL_VARIABLE:
            /* If a buffer is associated then we pick it up.           */
            if (GnlNodeHook (Node) && (GnlMapNodeInfoOriginalCompo (Node)))
            {
                UserCompo = GnlMapNodeInfoOriginalCompo (Node);
                if (BListAddElt (GnlComponents (Gnl), (int)UserCompo))
                    return (GNL_MEMORY_FULL);
                return (GNL_OK);
            }

            Var = (GNL_VAR) GnlNodeSons (Node);
            if (GnlCreateNodeForVar (TempGnl, Var, NewNode))

```

```

        return (GNL_MEMORY_FULL);
    return (GNL_OK);

case GNL_CONSTANTE:
    if (GnlCreateNode (TempGnl, GNL_CONSTANTE, NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*NewNode, GnlNodeSons (Node));
    return (GNL_OK);

case GNL_NOT:
case GNL_AND:
case GNL_NAND:
case GNL_OR:
    Var = GnlMapNodeInfoCutVar (Node);
    if (Var != OutVar)
    {
        if (GnlCreateNodeForVar (TempGnl, Var, NewNode))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

Bdd = GnlMapNodeInfoBestBdd (Node);
if (!Bdd)
    return (GNL_OK);
BddPtr = GetBddPtrFromBdd (Bdd);
if ((Bdd == bdd_one ()) ||
    (Bdd == bdd_zero ()))
{
    if (GnlCreateNode (TempGnl, GNL_CONSTANTE, NewNode))
        return (GNL_MEMORY_FULL);
    if (Bdd == bdd_one ())
        SetGnlNodeSons (*NewNode, (BLIST)1);
    else
        SetGnlNodeSons (*NewNode, (BLIST)0);
    return (GNL_OK);
}

ListDeriveCells = LibBddInfoDeriveCells (BddPtr);

if (GnlMapNodeInfoBestCell (Node))
{
    /* Best cell already stored */ *
    BestCell = GnlMapNodeInfoBestCell (Node);
}
else
{
    /* We extract the derive cell with the best Area */
    GnlGetBestAreaCell (ListDeriveCells, Bdd, &BestCell,
                        &Area, &Depth);
}

if (LibDeriveCellBdd (BestCell) != Bdd)
{
    printf ("ERROR: bdd differents\n");
    exit (1);
}

```

gnlmap.c

```
    if (GnlReconstructMappedCell (Gnl, OutVar, BestCell,
                                 ListDeriveCells, Node))
        return (GNL_MEMORY_FULL);

    *NewNode = NULL;
    return (GNL_OK);

default:
    GnlError (9 /* Unknown node */);
    return (GNL_OK);
}

/*
 * GnlUpdateFormalTristateCompo
 */
/*
 * This procedure updates the fields 'Formal' of each association of the
 * tri-state interface by taking the corresponding pin name in the
 * LIBC cell.
 */
GNL_STATUS GnlUpdateFormalTristateCompo (TristateCompo)
    GNL_TRISTATE_COMPONENT      TristateCompo;
{
    LIBC_CELL              Cell;
    LIBC_PIN               Pins;
    LIBC_NAME_LIST         ListPinName;
    char                  *Output;
    LIBC_BOOL_OPR          Input;
    LIBC_BOOL_OPR          Select;
    GNL_ASSOC              Assoc;
    GNL_ASSOC              InputAssoc;
    GNL_ASSOC              OutputAssoc;
    GNL_ASSOC              SelectAssoc;
    LIBC_BOOL_OPR          PinFunction;

    Cell = GnlTriStateCompoInfoCell (TristateCompo);

    Input = GnlTriStateCompoInfoInput (TristateCompo);
    InputAssoc = GnlTriStateInputAssoc (TristateCompo);

    Output = GnlTriStateCompoInfoOutput (TristateCompo);
    OutputAssoc = GnlTriStateOutputAssoc (TristateCompo);

    Select = GnlTriStateCompoInfoEnable (TristateCompo);
    SelectAssoc = GnlTriStateSelectAssoc (TristateCompo);

    Pins = LibCellPins (Cell);

    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simpel name */

        Assoc = NULL;
        if (GnlNameIsPresent (LibNameListName (ListPinName),
                             Input))
            Assoc = InputAssoc;
```

gnlmap.c

```
else if (GnlNameIsPresent (LibNameListName (ListPinName),
                           Select))
    Assoc = SelectAssoc;
else
{
    /* It is may be an output. */                                     */
    PinFunction = LibPinFunction (Pins);
    if (PinFunction)
    {
        Assoc = OutputAssoc;
    }
}
if (Assoc)
    SetGnlAssocFormalPort (Assoc, LibNameListName (ListPinName));
}

return (GNL_OK);
}

/*-----*/
/* GnlUpdateFormalBufCompo                                         */
/*-----*/
/* This procedure updates the fields 'Formal' of each association of the */
/* buffer interface by taking the corresponding pin name in the LIBC   */
/* cell.                                                               */
/*-----*/
GNL_STATUS GnlUpdateFormalBufCompo (BufCompo)
    GNL_BUF_COMPONENT      BufCompo;
{
    LIBC_CELL              Cell;
    LIBC_PIN               Pins;
    LIBC_NAME_LIST          ListPinName;
    char                   *Output;
    LIBC_BOOL_OPR           Input;
    LIBC_BOOL_OPR           Select;
    GNL_ASSOC               Assoc;
    GNL_ASSOC               InputAssoc;
    GNL_ASSOC               OutputAssoc;
    GNL_ASSOC               SelectAssoc;
    LIBC_BOOL_OPR           PinFunction;

    Cell = GnlBufCompoInfoCell (BufCompo);

    Input = GnlBufCompoInfoInput (BufCompo);
    InputAssoc = GnlBufInputAssoc (BufCompo);

    Output = GnlBufCompoInfoOutput (BufCompo);
    OutputAssoc = GnlBufOutputAssoc (BufCompo);

    Pins = LibCellPins (Cell);

    for (; Pins != NULL; Pins = LibPinNext (Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simpel name */
```

gnlmap.c

```
Assoc = NULL;
if (GnlNameIsPresent (LibNameListName (ListPinName),
                      Input))
    Assoc = InputAssoc;
else if (GnlNameIsPresent (LibNameListName (ListPinName),
                           Select))
    Assoc = SelectAssoc;
else
{
    /* It is may be an output. */
    PinFunction = LibPinFunction (Pins);
    if (PinFunction)
    {
        Assoc = OutputAssoc;
    }
}
if (Assoc)
    SetGnlAssocFormalPort (Assoc, LibNameListName (ListPinName));
}

return (GNL_OK);
}

/*
 * GnlUpdateFormalSeqCompo
 */
/*
 * This procedure updates the fields 'Formal' of each association of the
 * sequential compo. interface by taking the corresponding pin name in
 * the LIBC cell.
 */
GNL_STATUS GnlUpdateFormalSeqCompo (SeqCompo)
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
{
    LIBC_CELL          Cell;
    LIBC_PIN           Pins;
    LIBC_NAME_LIST     ListPinName;
    char               *Output;
    char               *OutputBar;
    LIBC_BOOL_OPR      Input;
    LIBC_BOOL_OPR      Clock;
    LIBC_BOOL_OPR      Clear;
    LIBC_BOOL_OPR      Preset;
    GNL_ASSOC          Assoc;
    GNL_ASSOC          InputAssoc;
    GNL_ASSOC          OutputAssoc;
    GNL_ASSOC          OutputBarAssoc;
    GNL_ASSOC          ClockAssoc;
    GNL_ASSOC          ClearAssoc;
    GNL_ASSOC          PresetAssoc;
    LIBC_BOOL_OPR      PinFunction;
    GNL_FORM_OUTPUT    FormOutput;
    char               *IQ;
    char               *IQN;
```

gnlmap.c

```
Cell = GnlSeqCompoInfoCell (SeqCompo);

IQ = LibFFLatchQName (LibCellFFLatch (Cell));
IQN = LibFFLatchQNNName (LibCellFFLatch (Cell));

FormOutput = GnlSequentialCompoFormOutput (SeqCompo);

Input = GnlSeqCompoInfoInput (SeqCompo);
InputAssoc = GnlSequentialCompoInputAssoc (SeqCompo);

Output = GnlSeqCompoInfoOutput (SeqCompo);
OutputAssoc = GnlSequentialCompoOutputAssoc (SeqCompo);

OutputBar = GnlSeqCompoInfoOutputBar (SeqCompo);
OutputBarAssoc = GnlSequentialCompoOutputBarAssoc (SeqCompo);

Clock = GnlSeqCompoInfoClock (SeqCompo);
ClockAssoc = GnlSequentialCompoClockAssoc (SeqCompo);

Clear = GnlSeqCompoInfoReset (SeqCompo);
ClearAssoc = GnlSequentialCompoResetAssoc (SeqCompo);

Preset = GnlSeqCompoInfoSet (SeqCompo);
PresetAssoc = GnlSequentialCompoSetAssoc (SeqCompo);

Pins = LibCellPins (Cell);

for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    ListPinName = LibPinName (Pins); /* we know it is a simpel name */

    Assoc = NULL;
    if (GnlNameIsPresent (LibNameListName (ListPinName),
                          Input))
        Assoc = InputAssoc;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),
                               Clock))
        Assoc = ClockAssoc;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),
                               Clear))
        Assoc = ClearAssoc;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),
                               Preset))
        Assoc = PresetAssoc;
    else
    {
        /* It is may be an output. */                                     */
        PinFunction = LibPinFunction (Pins);
        if (PinFunction)
        {
            if (GnlNameIsPresent (IQ, PinFunction))
                Assoc = OutputAssoc;
            else if (GnlNameIsPresent (IQN, PinFunction))
                Assoc = OutputBarAssoc;
        }
    }
}
```

gnlmap.c

```
if (Assoc)
    SetGnlAssocFormalPort (Assoc, LibNameListName (ListPinName));
}

return (GNL_OK);
}

/*
 * GnlUpdateFormalPorts
 */
/*
 * This procedure updates the fields 'Formal' of each association of the
 * current component interface by taking the corresponding pin name in
 * the LIBC cell.
 * We do that only for predefined components.
 */
GNL_STATUS GnlUpdateFormalPorts (Compo)
    GNL_COMPONENT      Compo;
{



switch (GnlComponentType (Compo)) {
    case GNL_SEQUENTIAL_COMPO:
        return (GnlUpdateFormalSeqCompo (
            (GNL_SEQUENTIAL_COMPONENT) Compo));

    case GNL_TRISTATE_COMPO:
        return (GnlUpdateFormalTristateCompo (
            (GNL_TRISTATE_COMPONENT) Compo));

    case GNL_BUF_COMPO:
        return (GnlUpdateFormalBufCompo (
            (GNL_BUF_COMPONENT) Compo));

    default:
        break;
}

return (GNL_OK);
}

/*
 * GnlFreeMapNodeInfo
 */
/*
 * Frees the structure pointed by 'GNL_TRISTATE_COMPO_INFO'
 */
void GnlFreeMapNodeInfo (Node)
    GNL_NODE Node;
{

if (GnlMapNodeInfoNodeSupport (Node))
    BListQuickDelete (&GnlMapNodeInfoNodeSupport (Node));
if (GnlMapNodeInfoLibVarSupport (Node))
    BListQuickDelete (&GnlMapNodeInfoLibVarSupport (Node));
}
```

gnlmap.c

```
if (GnlMapNodeInfoSNodes (Node))
    GnlFreeMapSNodes (GnlMapNodeInfoSNodes (Node));
free ((GNL_MAP_NODE_INFO) GnlNodeHook (Node));
}

/*
 *-----*
 * GnlCleanNodeInfo
 *-----*
 * This procedure cleans the Hook field of the node 'Node' which
 * corresponds to a 'GNL_MAP_NODE_INFO' structure
 *-----*/
void GnlCleanNodeInfo (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        if (GnlNodeHook (Node))
        {
            GnlFreeMapNodeInfo (Node);
            SetGnlNodeHook (Node, NULL);
        }
        return;
    }

    if (GnlNodeHook (Node))
    {
        GnlFreeMapNodeInfo (Node);
        SetGnlNodeHook (Node, NULL);
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE) BListElts (GnlNodeSons (Node), i);
        GnlCleanNodeInfo (SonI);
    }
}

/*
 *-----*
 * GnlUpdateFormalFieldOfPredefinedComponents
 *-----*
 * We scan the components and eventually update the formal field of each
 * association of SEQUENTIAL, TRISTATE or BUF components.
 *-----*/
GNL_STATUS GnlUpdateFormalFieldOfPredefinedComponents (Gnl)
    GNL      Gnl;
{
    BLIST      Components;
```

gnlmap.c

```
int i;
GNL_COMPONENT ComponentI;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT) BListElt (Components, i);
    if (GnlUpdateFormalPorts (ComponentI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----*/
/* GnlReconstructMappedGnl */
/*-----*/
/* This procedure analyzes a fresh mapped Gnl and replaces mapped */
/* GNL_NODE by new components corresponding to the libc cells. */
/*-----*/
GNL_STATUS GnlReconstructMappedGnl (Gnl, GnlLib)
    GNL          Gnl;
    GNL_LIB      GnlLib;
{
    int           IndexList;
    int           i;
    GNL_VAR       VarI;
    GNL_FUNCTION  FunctionI;
    LIB_CELL     BestInv;
    GNL          TempGnl;
    BLIST         NodesSegments;
    GNL_NODE     NewNode;
    GNL_COMPONENT ComponentI;
    BLIST         Components;

IndexList = 0;

BestInv = GetBestAreaLibInverter (GnlLib);

/* We create a temporary Gnl in which we will create new functions */
/* corresponding to simple variable assignment of the form : */
/* 'assign x = y;' */
/* We will then replace the GNL_NODE segments of 'Gnl' by the ones of */
/* 'GnlTemp' and remove the segments of 'Gnl'. */
if ((TempGnl = (GNL) malloc (1, sizeof(GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

SetGnlNbIn (TempGnl, GnlNbIn (Gnl));
SetGnlNbOut (TempGnl, GnlNbOut (Gnl));
SetGnlNbLocal (TempGnl, GnlNbLocal (Gnl));

SetGnlInputs (TempGnl, GnlInputs (Gnl));
SetGnlOutputs (TempGnl, GnlOutputs (Gnl));
SetGnlLocals (TempGnl, GnlLocals (Gnl));
```

gnlmap.c

```
SetGnlHashNames (TempGnl, GnlHashNames (Gnl)) ;

if (BListCreate (&NodesSegments))
    return (GNL_MEMORY_FULL) ;
SetGnlNodesSegments (TempGnl, NodesSegments) ;
SetGnlFirstNode (TempGnl, NULL) ;
SetGnlLastNode (TempGnl, NULL) ;

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);

    FunctionI = GnlVarFunction (Vari);

    if (GnlReconstructMappedFunction (Gnl, TempGnl, BestInv, Vari,
                                    GnlFunctionOnSet (FunctionI),
                                    &NewNode))
        return (GNL_MEMORY_FULL);

    if (NewNode)
    {
        SetGnlFunctionOnSet (FunctionI, NewNode);
        continue;
    }

    GnlFunctionFree (FunctionI);

    /* Vari has no function anymore but is the output of a user      */
    /* component.                                                 */
    GnlResetVarFunction (Vari);
    BListDelInsert (GnlFunctions (Gnl), i+1);
    i--;
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);

    FunctionI = GnlVarFunction (Vari);
    GnlCleanNodeInfo (GnlFunctionOnSet (FunctionI));
}

GnlFreeNodesSegments (Gnl);

/* We move the segments of nodes from 'TempGnl' to 'Gnl'.          */
SetGnlNodesSegments (Gnl, GnlNodesSegments (TempGnl));
SetGnlFirstNode (Gnl, GnlFirstNode (TempGnl));
SetGnlLastNode (Gnl, GnlLastNode (TempGnl));

free ((char*)TempGnl);

/* We scan the components and eventually update the formal field   */
/* of each association of SEQUENTIAL, TRISTATE or BUF components. */
/* if (GnlUpdateFormalFieldOfPredefinedComponents (Gnl))           */
    return (GNL_MEMORY_FULL);
```

gnlmap.c

```
    return (GNL_OK) ;
}

/*-----
/* GnlUpdateFormalFieldOfPredefinedComponentsRec          */
/*-----*/
GNL_STATUS GnlUpdateFormalFieldOfPredefinedComponentsRec (Nw, Gnl)
    GNL_NETWORK      Nw;
    GNL             Gnl;
{
    int                  i;
    BLIST                Components;
    GNL_COMPONENT        ComponentI;
    GNL_USER_COMPONENT   UserCompoI;
    GNL                 GnlCompoI;

    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));

    /* We scan the components and eventually update the formal field      */
    /* of each association of SEQUENTIAL, TRISTATE or BUF components.      */
    if (GnlUpdateFormalFieldOfPredefinedComponents (Gnl))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT) ComponentI;
        if (!GnlUserComponentGnlDef (UserCompoI))
            continue;

        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

        if (GnlUpdateFormalFieldOfPredefinedComponentsRec (Nw, GnlCompoI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlUpdateFormalFieldOfPredefinedComponentsInNw          */
/*-----*/
GNL_STATUS GnlUpdateFormalFieldOfPredefinedComponentsInNw (Nw)
    GNL_NETWORK      Nw;
{
    GNL             TopGnl;
```

gnlmap.c

```
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
TopGnl = GnlNetworkTopGnl (Nw);

if (GnlUpdateFormalFieldOfPredefinedComponentsRec (Nw, TopGnl))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* GnlGetReplaceVar
*/
GNL_VAR GnlGetReplaceVar (Var)
    GNL_VAR Var;
{
    GNL_VAR ReplaceVar;
    GNL_FUNCTION Function;
    GNL_NODE Node;

    if (!GnlVarFunction (Var))
        return (Var);

    Function = GnlVarFunction (Var);
    Node = GnlFunctionOnSet (Function);

    if (GnlNodeOp (Node) != GNL_VARIABLE)
        return (Var);

    ReplaceVar = (GNL_VAR) GnlNodeSons (Node);
    return (GnlGetReplaceVar (ReplaceVar));
}

/*-----
/* GnlDeleteFunctionRec
*/
void GnlDeleteFunctionRec (Var)
    GNL_VAR Var;
{
    GNL_VAR ReplaceVar;
    GNL_FUNCTION Function;
    GNL_NODE Node;

    if (!GnlVarFunction (Var))
        return;

    Function = GnlVarFunction (Var);
    Node = GnlFunctionOnSet (Function);

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) != GNL_VARIABLE)
    {
```

gnlmap.c

```
    fprintf (stderr,
              " ERROR: non simple Boolean equation detected\n");
    exit (1);
}

ReplaceVar = (GNL_VAR)GnlNodeSons (Node);

GnlDeleteFunctionRec (ReplaceVar);

GnlFunctionFree (Function);
SetGnlVarFunction (Var, NULL);
}

/*-----*/
/* GnlReplaceVarInNode */
/*-----*/
GNL_STATUS GnlReplaceVarInNode (Node, ListReplaceVar)
    GNL_NODE Node;
    BLIST ListReplaceVar;
{
    int i;
    GNL_NODE SonI;
    GNL_VAR Var;
    GNL_VAR ReplaceVar;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        ReplaceVar = GnlGetReplaceVar (Var);
        if (Var != ReplaceVar)
        {
            if (BListAddElt (ListReplaceVar, (int)Var))
                return (GNL_MEMORY_FULL);
            SetGnlNodeSons (Node, (BLIST)ReplaceVar);
        }
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlReplaceVarInNode (SonI, ListReplaceVar))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlReplaceMappedSimpleSignals */
/*-----*/
```

gnlmap.c

```
/* This procedure analyzes each actual var of the interfaces of all the */
/* components and replace them if they are simple signals assigned. A      */
/* simple signal assigned is the following case:                           */
/*          assign x = y;                                              */
/*          assign y = z;                                              */
/*          toto u1 (.z(x), .a(b));                                     */
/*          */                                                       */
/* The goal above is the replace 'x' used as actual signal in the      */
/* component 'toto' and replace it by x --> y --> z                  */
/*-----*/
```

GNL_STATUS GnlReplaceMappedSimpleSignals (Gnl)

```
    GNL      Gnl;
{
    BLIST          Components;
    int             i;
    int             j;
    int             k;
    GNL_COMPONENT   ComponentI;
    GNL_SEQUENTIAL_COMPONENT SeqCompoI;
    GNL_TRISTATE_COMPONENT TriCompoI;
    GNL_USER_COMPONENT UserCompoI;
    GNL_BUF_COMPONENT BufCompoI;
    BLIST          Interface;
    GNL_VAR         Actual;
    GNL_ASSOC       AssocK;
    BLIST          ListSignals;
    GNL_VAR         VarJ;
    GNL_VAR         ReplaceVar;
    GNL_VAR         Input;
    GNL_VAR         Formal;
    BLIST          ListReplacedVar;
    GNL_VAR         VarI;
    BLIST          HashTableNames;
    BLIST          BucketI;
    GNL_FUNCTION    Function;
    GNL_NODE        Node;

    /* No simplification if we are running in NL delay mode.           */
    if (GnlEnvCriterion () == GNL_NL_DELAY)
        return (GNL_OK);

    if (BListCreate (&ListReplacedVar))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);
        switch (GnlComponentType (ComponentI)) {
            case GNL_SEQUENTIAL_COMPO:
                SeqCompoI = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
                Input = GnlSequentialCompoInput (SeqCompoI);
                ReplaceVar = GnlGetReplaceVar (Input);
                if (ReplaceVar != Input)
                {
                    SetGnlSequentialCompoInput (SeqCompoI,
```

```

                    ReplaceVar);
        if (BListAddElt (ListReplacedVar, (int) Input))
            return (GNL_MEMORY_FULL);
    }
    continue;
    break;

case GNL_TRISTATE_COMPO:
    TriCompoI = (GNL_TRISTATE_COMPONENT) ComponentI;
    Input = GnlTriStateInput (TriCompoI);
    ReplaceVar = GnlGetReplaceVar (Input);
    if (ReplaceVar != Input)
    {
        SetGnlTriStateInput (TriCompoI, ReplaceVar);
        if (BListAddElt (ListReplacedVar, (int) Input))
            return (GNL_MEMORY_FULL);
    }
    continue;
    break;

case GNL_BUF_COMPO:
    BufCompoI = (GNL_BUF_COMPONENT) ComponentI;
    Input = GnlBufInput (BufCompoI);
    ReplaceVar = GnlGetReplaceVar (Input);
    if (ReplaceVar != Input)
    {
        SetGnlBufInput (BufCompoI, ReplaceVar);
        if (BListAddElt (ListReplacedVar, (int) Input))
            return (GNL_MEMORY_FULL);
    }
    continue;
    break;

case GNL_USER_COMPO:
    UserCompoI = (GNL_USER_COMPONENT) ComponentI;
    /* It is apparently a black box so we do not do      */
    /* anything.                                         */
    if (GnlUserComponentFormalType (UserCompoI) !=      *
        GNL_FORMAL_CHAR)
        continue;

    Interface = GnlUserComponentInterface (UserCompoI);
    break;

default:
    fprintf (stderr, " ERROR: unknown component\n");
    exit (1);
}

/* In the case of 'GNL_USER_COMPO' ...
for (k=0; k<BListSize (Interface); k++)
{
    AssocK = (GNL_ASSOC) BListElt (Interface, k);
    Formal = GnlAssocFormalPort (AssocK);
    Actual = GnlAssocActualPort (AssocK);

    /* If it is an open port we do nothing           */

```

```

    if (!Actual)
        continue;

    if (GnlVarIsVar (Actual))
    {
        ReplaceVar = GnlGetReplaceVar (Actual);
        if (ReplaceVar != Actual)
        {
            SetGnlAssocActualPort (AssocK, ReplaceVar);
            if (BListAddElt (ListReplacedVar, (int)Actual))
                return (GNL_MEMORY_FULL);
        }
        continue;
    }

/* Otherwise it is a GNL_NODE with GNL_CONCAT operator */
ListSignals = GnlNodeSons ((GNL_NODE)Actual);
for (j=0; j<BListSize (ListSignals); j++)
{
    VarJ = (GNL_VAR)BListElt (ListSignals, j);
    ReplaceVar = GnlGetReplaceVar (VarJ);
    if (ReplaceVar != VarJ)
    {
        BListElt (ListSignals, j) = (int)ReplaceVar;
        if (BListAddElt (ListReplacedVar, (int)VarJ))
            return (GNL_MEMORY_FULL);
    }
}
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    Function = GnlVarFunction (VarI);
    Node = GnlFunctionOnSet (Function);
    if (GnlReplaceVarInNode (Node, ListReplacedVar))
        return (GNL_MEMORY_FULL);
}

/* We analyze now the list 'ListReplacedVar' in order to destroy the */
/* equations associated to them.                                         */
for (i=0; i<BListSize (ListReplacedVar); i++)
{
    VarI = (GNL_VAR)BListElt (ListReplacedVar, i);
    GnlDeleteFunctionRec (VarI);
}
BListQuickDelete (&ListReplacedVar);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    if (!GnlVarFunction (VarI))
    {
        BListDelInsert (GnlFunctions (Gnl), i+1);
        i--;
    }
}

```

gnlmap.c

```
        }
    }

    return (GNL_OK);
}

/*-----*/
/* GnlMap */
/*-----*/
/* General technology mapping function which takes as input a Gnl and a */
/* specific library. Purpose is to map the Generic Gnl in term of compo-*/
/* nents of the specific library 'GnlLib'. */
/* 'Done' is 1 if the mapping was done and 0 is the library was not */
/* powerfull enough to map the global Gnl. */
/* A Verilog dumping of the result of the mapping is done in the file */
/* 'OutFile'. */
/*-----*/
GNL_STATUS GnlMap (Nw, Gnl, GnlLibc, OutFile, Effort, Criter, Done)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    LIBC_LIB GnlLibc;
    FILE            *OutFile;
    int              Effort;           /* 0 --> Low, 1 --> High          */
    int              Criter;
    int              *Done;

{
    double   TechArea;
    int      TechDepth;
    float   TechMax;
    float   TechMin;
    int      Tag;
    int      NbOriginalFunctions;
    GNL_LIB  GnlLib;

    /* Accessing our private library description. */
    GnlLib = (GNL_LIB)LibHook (GnlLibc);

    *Done = 1;
    G_Effort = Effort;
    Tag = GnlTag (Gnl);

    if (!GnlEnvDontTouch ())
    {
        if (GnlInjectSingleVarWithMaxLitt (Gnl, 1))
            return (GNL_MEMORY_FULL);

        /* In case of NL delay and touch netlist we do an important */
        /* collapsing. */
        if (GnlEnvCriterion () == GNL_NL_DELAY)
        {
            if (GnlInjectQuick (Gnl, 100))
                return (GNL_MEMORY_FULL);
        }
        else
    }
```

gnlmap.c

```
        if (GnlInjectQuick (Gnl, 2))
            return (GNL_MEMORY_FULL);
    }

/* We sort the functions according to their level. First one is the */
/* highest, last function is the deepest.                                */
if (BListSize (GnlFunctions (Gnl)))
{
    fprintf (stderr, " Sorting functions...\n");
    GnlSortFunctionsOnLevel (Gnl);
}

if (!GnlEnvDontTouch ())
{
    /* We transform the Gnl as a tree of AND nodes                      */
    if (GnlEnvCriterion () == GNL_NL_DELAY)
    {
        if (GnlAndifyPropagate (Gnl))
            return (GNL_MEMORY_FULL);
        GnlSortFunctionsOnInvertLevel (Gnl);
        if (GnlMakeBalanceBinary (Gnl, Criter))
            return (GNL_MEMORY_FULL);
    }
    else if (G_Effort == 1)
    {
        if (GnlAndify (Gnl))
            return (GNL_MEMORY_FULL);
        GnlSortFunctionsOnInvertLevel (Gnl);
        if (GnlMakeBalanceBinary (Gnl, Criter))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlAndifyPropagate (Gnl))
            return (GNL_MEMORY_FULL);
        GnlSortFunctionsOnInvertLevel (Gnl);
        if (GnlMakeBalanceBinary (Gnl, Criter))
            return (GNL_MEMORY_FULL);
    }
}
else
{
    /* we add a couple of inverters at each outputs.                      */
    if (GnlAddOutputsCoupleInverter (Gnl))
        return (GNL_MEMORY_FULL);
}

if (GnlAndify (Gnl))
    return (GNL_MEMORY_FULL);
GnlSortFunctionsOnInvertLevel (Gnl);
if (GnlMakeBalanceBinary (Gnl, Criter))
    return (GNL_MEMORY_FULL);

/* we reset the Hook field of each var                                 */
GnlResetVarHook (Gnl);
```

gnlmap.c

```
fprintf (stderr, "\n ** JUST BEFORE MAPPING **\n");
GnlPrintGnl (stderr, Gnl);

fprintf (stderr, "\n Technology Mapping:\n");
NbOriginalFunctions = BListSize (GnlFunctions (Gnl));
if (GnlMapGnl (Nw, Gnl, GnlLibc, G_Effort, Done))
    return (GNL_MEMORY_FULL);

if (!(*Done))
{
    fprintf (stderr,
        " ERROR: cannot map the whole Boolean network\n");
    return (GNL_OK);
}

fprintf (stderr, " Sorting Gates...\n");
GnlSortFunctionsOnLevel (Gnl);

if (GnlCreateCutVars (Gnl))
    return (GNL_MEMORY_FULL);

if (GnlEnvCriterion () == GNL_AREA)
{
    fprintf (stderr, " Gates compaction...\n");
    if (GnlFactorizeGates (Gnl, NbOriginalFunctions))
        return (GNL_MEMORY_FULL);
}

/* Creates inverters for BDDs pointed as complement */ 
if (GnlCreateInverterGate (Gnl, GnlLib))
    return (GNL_MEMORY_FULL);

TechArea = GnlGetGnlMapArea (Gnl);
TechDepth = GnlGetMaxTechnologyDepth (Gnl);

SetGnlArea (Gnl, TechArea);
SetGnlDepth (Gnl, TechDepth);
SetGnlTechMax (Gnl, TechMax);
SetGnlTechMin (Gnl, TechMin);

/* We replace mapped GNL_NODE by new components corresponding to the */
/* libc cells. */ 
if (GnlReconstructMappedGnl (Gnl, GnlLib))
    return (GNL_MEMORY_FULL);

/* We replace simple signals by their corresponding one */ 
if (GnlReplaceMappedSimpleSignals (Gnl))
    return (GNL_MEMORY_FULL);

/* This function computes the Max Fanout in the gnl 'Gnl' and stores */
/* it in the field 'GnlMaxFanout (Gnl)'. */ 
GnlComputeMaxFanoutInGnl (Gnl);

fprintf (stderr, "\n Extracted Data After Technology Mapping:\n");
fprintf (stderr, "          o Total Gates Area      = ");
fprintf (stderr, "%4.f units\n", TechArea);
```

gnlmap.c

```
fprintf (stderr, "      o Max. Comb. Wire Path = %d wires\n",
TechDepth);
fprintf (stderr, "      o Max. Fanout           = %d nets\n",
GnlMaxFanout (Gnl));

fprintf (stderr, "\n");

#ifndef STAT
fprintf (stderr, "# G_NB_TARGET_FUNC = %d\n", G_NB_TARGET_FUNC);
fprintf (stderr, "# G_ROBDD_BUILT = %d\n", G_ROBDD_BUILT);
fprintf (stderr, "# G_NB_CELL_FOUND = %d\n", G_NB_CELL_FOUND);
#endif

SetGnlTag (Gnl, Tag);
return (GNL_OK);
}

/*----- EOF -----*/
```

gnlmap.h

```
/*-----*/
/*
 *      File:          gnlmap.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*-----*/
#ifndef GNLMAP_H
#define GNLMAP_H

/*-----*/
/* GNL_SNODE */
/*-----*/
typedef struct GNL_SNODE_STRUCT {
    GNL_NODE           Node;
    struct GNL_SNODE_STRUCT *Next;
}
GNL_SNODE_REC, *GNL_SNODE;

#define GnlSNodeNode(n)      ((n)->Node)
#define GnlSNodeNext(n)     ((n)->Next)

#define SetGnlSNodeNode(n,s) ((n)->Node = s)
#define SetGnlSNodeNext(n,s) ((n)->Next = s)

/*-----*/
/* Info stored on GNL_SEQUENTIAL_COMPONENT thru Hook field */
/*-----*/
typedef struct GNL_SEQUENTIAL_COMPO_INFO_STRUCT {
    LIBC_CELL           Cell;
    LIBC_BOOL_OPR      Input;
    char                *Output;
    char                *OutputBar;
    LIBC_BOOL_OPR      Clock;
    LIBC_BOOL_OPR      Reset;
    LIBC_BOOL_OPR      Set;
    LIBC_BOOL_OPR      Enable;
    void               *Hook;
}
GNL_SEQUENTIAL_COMPO_INFO_REC, *GNL_SEQUENTIAL_COMPO_INFO;

#define GnlSeqCompoInfoCell(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Cell)
#define GnlSeqCompoInfoInput(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Input)
#define GnlSeqCompoInfoOutput(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Output)
#define GnlSeqCompoInfoOutputBar(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->OutputBar)
#define GnlSeqCompoInfoClock(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Clock)
#define GnlSeqCompoInfoReset(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Reset)
#define GnlSeqCompoInfoSet(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Set)
```

gnlmap.h

```
#define GnlSeqCompoInfoHook(n) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Hook)

#define SetGnlSeqCompoInfoCell(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Cell = c)
#define SetGnlSeqCompoInfoInput(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Input = c)
#define SetGnlSeqCompoInfoOutput(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Output = c)
#define SetGnlSeqCompoInfoOutputBar(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->OutputBar = c)
#define SetGnlSeqCompoInfoClock(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Clock = c)
#define SetGnlSeqCompoInfoReset(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Reset = c)
#define SetGnlSeqCompoInfoSet(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Set = c)
#define SetGnlSeqCompoInfoHook(n,c) \
    (((GNL_SEQUENTIAL_COMPO_INFO)((n)->Hook))->Hook = c)

/*-----*/
/* Info stored on GNL_TRISTATE_COMPONENT thru Hook field */
/*-----*/
typedef struct GNL_TRISTATE_COMPO_INFO_STRUCT {
    LIBC_CELL           Cell;
    LIBC_BOOL_OPR       Input;
    char                *Output;
    LIBC_BOOL_OPR       Enable;
    void                *Hook;
} GNL_TRISTATE_COMPO_INFO_REC, *GNL_TRISTATE_COMPO_INFO;

#define GnlTriStateCompoInfoCell(n) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Cell)
#define GnlTriStateCompoInfoInput(n) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Input)
#define GnlTriStateCompoInfoOutput(n) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Output)
#define GnlTriStateCompoInfoEnable(n) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Enable)
#define GnlTriStateCompoInfoHook(n) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Hook)

#define SetGnlTriStateCompoInfoCell(n,c) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Cell = c)
#define SetGnlTriStateCompoInfoInput(n,c) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Input = c)
#define SetGnlTriStateCompoInfoOutput(n,c) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Output = c)
#define SetGnlTriStateCompoInfoEnable(n,c) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Enable = c)
#define SetGnlTriStateCompoInfoHook(n,c) \
    (((GNL_TRISTATE_COMPO_INFO)((n)->Hook))->Hook = c)

/*-----*/
/* Info stored on GNL_BUF_COMPONENT thru Hook field */
/*-----*/
```

gnlmap.h

```
/*-----*/
typedef struct GNL_BUF_COMPO_INFO_STRUCT {
    LIBC_CELL           Cell;
    LIBC_BOOL_OPR      Input;
    char               *Output;
    void               *Hook;
}
GNL_BUF_COMPO_INFO_REC, *GNL_BUF_COMPO_INFO;

#define GnlBufCompoInfoCell(n) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Cell)
#define GnlBufCompoInfoInput(n) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Input)
#define GnlBufCompoInfoOutput(n) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Output)
#define GnlBufCompoInfoHook(n) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Hook)

#define SetGnlBufCompoInfoCell(n,c) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Cell = c)
#define SetGnlBufCompoInfoInput(n,c) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Input = c)
#define SetGnlBufCompoInfoOutput(n,c) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Output = c)
#define SetGnlBufCompoInfoHook(n,c) \
    (((GNL_BUF_COMPO_INFO)((n)->Hook))->Hook = c)

/*-----*/
/* Info stored on GNL_NODE thru Hook field */
/*-----*/
typedef struct GNL_MAP_NODE_INFO_STRUCT {
    BDD                Bdd;
    GNL_VAR            CutVar;
    BDD                BestBdd;
    BLIST              NodeSupport;
    BLIST              LibVarSupport;
    GNL_VAR            LibraryVar;
    GNL_SNODE          SNodes;

    GNL_USER_COMPONENT OriginalCompo;
    LIB_DERIVE_CELL    BestCell;
    float              BestArea;
    float              BestPower;
    float              BestArrivalR;
    float              BestArrivalF;
    float              TransitionR;
    float              TransitionF;
    int                Depth;
    float              ATime;
    int                BestDepth;
    int                BestNet;
}
GNL_MAP_NODE_INFO_REC, *GNL_MAP_NODE_INFO;

/*
#define GnlMapNodeInfoOriginalCompo(n) \

```

gnlmap.h

```
(((GNL_NODE_INFO)((n)->Hook))->OriginalCompo)
*/
#define GnlMapNodeInfoBestCell(n)      \
    (((GNL_NODE_INFO)((n)->Hook))->BestCell)
#define GnlMapNodeInfoBdd(n)          \
    (((GNL_NODE_INFO)((n)->Hook))->Bdd)
#define GnlMapNodeInfoCutVar(n)       \
    (((GNL_NODE_INFO)((n)->Hook))->CutVar)
#define GnlMapNodeInfoBestArea(n)     \
    (((GNL_NODE_INFO)((n)->Hook))->BestArea)
#define GnlMapNodeInfoBestPower(n)    \
    (((GNL_NODE_INFO)((n)->Hook))->BestPower)
#define GnlMapNodeInfoBestArrivalR(n) \
    (((GNL_NODE_INFO)((n)->Hook))->BestArrivalR)
#define GnlMapNodeInfoBestArrivalF(n) \
    (((GNL_NODE_INFO)((n)->Hook))->BestArrivalF)
#define GnlMapNodeInfoTransitionR(n)  \
    (((GNL_NODE_INFO)((n)->Hook))->TransitionR)
#define GnlMapNodeInfoTransitionF(n)  \
    (((GNL_NODE_INFO)((n)->Hook))->TransitionF)
#define GnlMapNodeInfoBestBdd(n)      \
    (((GNL_NODE_INFO)((n)->Hook))->BestBdd)
#define GnlMapNodeInfoNodeSupport(n)   \
    (((GNL_NODE_INFO)((n)->Hook))->NodeSupport)
#define GnlMapNodeInfoLibVarSupport(n) \
    (((GNL_NODE_INFO)((n)->Hook))->LibVarSupport)
#define GnlMapNodeInfoLibVar(n)        \
    (((GNL_NODE_INFO)((n)->Hook))->LibraryVar)
#define GnlMapNodeInfoSNodes(n)        \
    (((GNL_NODE_INFO)((n)->Hook))->SNodes)
#define GnlMapNodeInfoDepth(n)         \
    (((GNL_NODE_INFO)((n)->Hook))->Depth)
#define GnlMapNodeInfoBestDepth(n)    \
    (((GNL_NODE_INFO)((n)->Hook))->BestDepth)
#define GnlMapNodeInfoBestNet(n)      \
    (((GNL_NODE_INFO)((n)->Hook))->BestNet)
#define GnlMapNodeInfoAtime(n)        \
    (((GNL_NODE_INFO)((n)->Hook))->Atime)

/*
#define SetGnlMapNodeInfoOriginalCompo(n,b)      \
    (((GNL_NODE_INFO)((n)->Hook))->OriginalCompo = b)
*/
#define SetGnlMapNodeInfoBestCell(n,b)      \
    (((GNL_NODE_INFO)((n)->Hook))->BestCell = b)
#define SetGnlMapNodeInfoBdd(n,b)          \
    (((GNL_NODE_INFO)((n)->Hook))->Bdd = b)
#define SetGnlMapNodeInfoCutVar(n,c)       \
    (((GNL_NODE_INFO)((n)->Hook))->CutVar = c)
#define SetGnlMapNodeInfoBestArea(n,c)     \
    (((GNL_NODE_INFO)((n)->Hook))->BestArea = c)
#define SetGnlMapNodeInfoBestPower(n,c)    \
    (((GNL_NODE_INFO)((n)->Hook))->BestPower = c)
#define SetGnlMapNodeInfoBestArrivalR(n,c) \
    (((GNL_NODE_INFO)((n)->Hook))->BestArrivalR = c)
#define SetGnlMapNodeInfoBestArrivalF(n,c) \
    (((GNL_NODE_INFO)((n)->Hook))->BestArrivalF = c)
```

gnlmap.h

```
#define SetGnlMapNodeInfoTransitionR(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->TransitionR = c)
#define SetGnlMapNodeInfoTransitionF(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->TransitionF = c)
#define SetGnlMapNodeInfoBestBdd(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->BestBdd = c)
#define SetGnlMapNodeInfoNodeSupport(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->NodeSupport = c)
#define SetGnlMapNodeInfoLibVarSupport(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->LibVarSupport = c)
#define SetGnlMapNodeInfoLibVar(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->LibraryVar = c)
#define SetGnlMapNodeInfoSNodes(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->SNodes = c)
#define SetGnlMapNodeInfoDepth(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->Depth = c)
#define SetGnlMapNodeInfoBestDepth(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->BestDepth = c)
#define SetGnlMapNodeInfoBestNet(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->BestNet = c)
#define SetGnlMapNodeInfoATime(n,c)      \
    (((GNL_MAP_NODE_INFO)((n)->Hook))->ATime = c)

/* ----- EOF ----- */
#endif
```

gnlmap3state.c

```
/*-----*/
/*
 *      File:          gnlmap3state.c
 *      Modifications: -
 *      Documentation: -
 *      Description:   */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlmap.h"

#include "blist.e"

/*-----*/
/* GnlCreateTriStateCompoInfo
/*-----*/
GNL_STATUS GnlCreateTriStateCompoInfo (NewTriStateCompoInfo)
    GNL_TRISTATE_COMPO_INFO      *NewTriStateCompoInfo;
{
    if ((*NewTriStateCompoInfo = (GNL_TRISTATE_COMPO_INFO)
        calloc (1, sizeof (GNL_TRISTATE_COMPO_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlExtractLibc3States
/*-----*/
int LibCellWith3StatePin (Cell, OutPin)
    LIBC_CELL              Cell;
    LIBC_PIN      *OutPin;
{
    LIBC_PIN      Pins;

    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext (Pins))
    {
```

gnlmap3state.c

```
if (LibPin3State (Pins))
{
    *OutPin = Pins;
    return (1);
}
}

return (0);
}

/*-----
/* Gnl3StateCellCanBeSupported
*/
-----*/
int Gnl3StateCellCanBeSupported (Cell)
    LIBC_CELL           Cell;
{
    LIBC_PIN             OutPin;
    LIBC_NAME_LIST       ListPinName;
    LIBC_BOOL_OPR        ThreeState;
    LIBC_BOOL_OPR        Function;

    if (!LibCellWith3StatePin (Cell, &OutPin))
        return (0);

    ListPinName = LibPinName (OutPin);
    if (!GnlSinglePinName (ListPinName))
    {
        fprintf (stderr,
            " WARNING: cell is ignored because of complex pin name: [%s]\n",
            LibCellName(Cell));
        return (0);
    }

    Function = LibPinFunction (OutPin);
    if (!GnlBoolOprIsSimpleFunction (Function))
    {
        fprintf (stderr,
            " WARNING: cell is ignored because of complex Input pin function: [%s]\n",
            LibCellName(Cell));
        return (0);
    }

    ThreeState = LibPin3State (OutPin);
    if (!GnlBoolOprIsSimpleFunction (ThreeState))
    {
        fprintf (stderr,
            " WARNING: cell is ignored because of complex Enable pin function: [%s]\n",
            LibCellName(Cell));
        return (0);
    }

    return (1);
}
```

gnlmap3state.c

```
}

/*-----*/
/* GnlExtractLibc3States */ */
/*-----*/
GNL_STATUS GnlExtractLibc3States (GnlLibc, Libc3States)
    LIBC_LIB      GnlLibc;
    BLIST        *Libc3States;
{
    LIBC_CELL      CellI;
    LIBC_FF_LATCH  FFLatch;
    LIBC_PIN       OutPin;

    if (BListCreate (Libc3States))
        return (GNL_MEMORY_FULL);

    CellI = LibCells (GnlLibc);
    for (; CellI != NULL; CellI = LibCellNext (CellI))
    {
        /* If sequential cell or cell not to use */
        if (LibCellFFLatch (CellI) || LibCellDontUse (CellI))
            continue;

        /* if cell with no 3 state pins then we continue. */
        if (!LibCellWith3StatePin (CellI, &OutPin))
            continue;

        if (!Gnl3StateCellCanBeSupported (CellI))
            continue;

        if (BListAddElt (*Libc3States, (int)CellI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlExtractGnl3States */ */
/*-----*/
GNL_STATUS GnlExtractGnl3States (Gnl, Gnl3States)
    GNL          Gnl;
    BLIST        *Gnl3States;
{
    BLIST          Components;
    int           j;
    GNL_COMPONENT ComponentJ;
    GNL_SEQUENTIAL_COMPONENT SeqCompoJ;

    if (BListCreate (Gnl3States))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
```

gnlmap3state.c

```
if (!Components)
    return (GNL_OK);

for (j=0; j<BListSize (Components); j++)
{
    ComponentJ = (GNL_COMPONENT)BListElt (Components, j);
    if (GnlComponentType (ComponentJ) != GNL_TRISTATE_COMPO)
        continue;

    if (BListAddElt (*Gnl3States, (int)ComponentJ))
        return (GNL_MEMORY_FULL);

}

return (GNL_OK);
}
```

```
/*
 * CmpLibcCellTriStateAreaPinMatch
 */
/*
 * Compare 'cell1' vs. 'cell2' by taking into account feature of the
 * current tristate component 'G_Current3StateComponent'. The ordering
 * is done in priority to take care about the polarity of the pin of
 * 'G_Current3StateComponent' which is 'select'.
 * The LIBC_CELL matching is the one with the correct the polarity in
 * priority.
 */
static GNL_TRISTATE_COMPONENT G_Current3StateComponent;

static int CmpLibcCellTriStateAreaPinMatch (Cell1, Cell2)
    LIBC_CELL    *Cell1;
    LIBC_CELL    *Cell2;
{
    LIBC_PIN      OutPin1;
    LIBC_PIN      OutPin2;
    LIBC_BOOL_OPR Function1;
    LIBC_BOOL_OPR Function2;

    if (!LibCellWith3StatePin (*Cell1, &OutPin1))
        return (-1);           /* should never happened ! */

    if (!LibCellWith3StatePin (*Cell2, &OutPin2))
        return (-1);           /* should never happened ! */

    Function1 = LibPinFunction (OutPin1);
    Function2 = LibPinFunction (OutPin2);

    if (GnlTriStateSelectPol (G_Current3StateComponent) == 1)
    {
        if (GnlBoolOprIsSimpleSignal (LibPin3State (OutPin1)) &&
            !GnlBoolOprIsSimpleSignal (LibPin3State (OutPin2)))
            return (1);
        if (!GnlBoolOprIsSimpleSignal (LibPin3State (OutPin1)) &&
            GnlBoolOprIsSimpleSignal (LibPin3State (OutPin2)))
            return (-1);
    }
}
```

gnlmap3state.c

```
        return (-1);
    }
else
{
    if (GnlBoolOprIsSimpleSignal (LibPin3State (OutPin1)) &&
        !GnlBoolOprIsSimpleSignal (LibPin3State (OutPin2)))
        return (-1);
    if (!GnlBoolOprIsSimpleSignal (LibPin3State (OutPin1)) &&
        GnlBoolOprIsSimpleSignal (LibPin3State (OutPin2)))
        return (1);
}

if (GnlTriStateInputPol (G_Current3StateComponent) == 1)
{
    if (GnlBoolOprIsSimpleSignal (Function1) &&
        !GnlBoolOprIsSimpleSignal (Function2))
        return (-1);
    if (!GnlBoolOprIsSimpleSignal (Function1) &&
        GnlBoolOprIsSimpleSignal (Function2))
        return (1);
}
else
{
    if (GnlBoolOprIsSimpleSignal (Function1) &&
        !GnlBoolOprIsSimpleSignal (Function2))
        return (1);
    if (!GnlBoolOprIsSimpleSignal (Function1) &&
        GnlBoolOprIsSimpleSignal (Function2))
        return (-1);
}

/* Everything is ok regarding the polarity of the pins so we decide */
/* with the area.                                                 */
/* We privilegeate the minimum area                                */
if (LibCellArea(*Cell1) < LibCellArea(*Cell2))
    return (-1);

if (LibCellArea(*Cell1) == LibCellArea(*Cell2))
    return (0);

return (1);
}

/*
 *-----*/
/* GnlConstruct3StateLogicBoundaries                               */
/*-----*/
GNL_STATUS GnlConstruct3StateLogicBoundaries (Gnl, TriState)
    GNL                      Gnl;
    GNL_TRISTATE_COMPONENT   TriState;
{
    char                  *OutPut;
    LIBC_BOOL_OPR          Input;
    LIBC_BOOL_OPR          Enable;
    GNL_VAR                InputVar;
    GNL_VAR                EnableVar;
    GNL_VAR                NewVar;
```

gnlmap3state.c

```
GNL_NODE      NewInputNode;
GNL_NODE      NewEnableNode;
GNL_FUNCTION  NewFunction;
GNL_NODE      VarNode;

OutPut = GnlTriStateCompoInfoOutput (TriState);
Input = GnlTriStateCompoInfoInput (TriState);
Enable = GnlTriStateCompoInfoEnable (TriState);

/*
GnlPrintBoolOpr (Input);
printf ("\n");
printf ("%s", OutPut);
printf ("\n");
GnlPrintBoolOpr (Enable);
printf ("\n");
*/
if ((!GnlBoolOprIsSimpleSignal (Input) &&
    GnlTriStateInputPol (TriState)) ||
    (GnlBoolOprIsSimpleSignal (Input) &&
     !GnlTriStateInputPol (TriState)))
{
    InputVar = GnlTriStateInput (TriState);

    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, InputVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewInputNode))
        return (GNL_MEMORY_FULL);

    /* Look if this expression already exists ... */
    if (GnlLookForSameExpression (Gnl, NewInputNode, &NewVar))
    {
        SetGnlTriStateInput (TriState, NewVar);
        if (!GnlVarIsPrimary (NewVar))
            SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
    else
    {
        if (GnlCreateUniqueVar (Gnl, GnlVarName (InputVar), &NewVar))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals(Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

        if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);

        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionVar (NewFunction, NewVar);
        SetGnlFunctionOnSet (NewFunction, NewInputNode);

        if (BListAddElt (GnlFunctions(Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
    }
}
```

gnlmap3state.c

```
        SetGnlTriStateInput (TriState, NewVar);
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
}

/* Gnl tristate are active for the transparent phase and LIBC */
/* tri-states are active for the Z phase. */
if ((GnlBoolOprIsSimpleSignal (Enable) &&
     GnlTriStateSelectPol (TriState)) ||
    (!GnlBoolOprIsSimpleSignal (Enable) &&
     !GnlTriStateSelectPol (TriState)))
{
    EnableVar = GnlTriStateSelect (TriState);

    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, EnableVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewEnableNode))
        return (GNL_MEMORY_FULL);

    /* Look if this expression already exists ... */
    if (GnlLookForSameExpression (Gnl, NewEnableNode, &NewVar))
    {
        SetGnlTriStateSelect (TriState, NewVar);
        if (!GnlVarIsPrimary (NewVar))
            SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
    else
    {
        if (GnlCreateUniqueVar (Gnl, GnlVarName (EnableVar), &NewVar))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

        if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);

        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionVar (NewFunction, NewVar);
        SetGnlFunctionOnSet (NewFunction, NewEnableNode);

        if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);

        SetGnlTriStateSelect (TriState, NewVar);
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlMap3STATE */
*/
```

gnlmap3state.c

```
/*-----*/
GNL_STATUS GnlMap3STATE (Gnl, TriState, LibcTriState)
    GNL          Gnl;
    GNL_TRISTATE_COMPONENT   TriState;
    BLIST         LibcTriState;
{
    int           i;
    LIBC_CELL     Cell;
    int           Match;
    GNL_STATUS    GnlStatus;
    LIBC_PIN      OutPin;
    LIBC_BOOL_OPR Function;
    LIBC_BOOL_OPR ThreeState;
    LIBC_NAME_LIST ListPinName;
    GNL_TRISTATE_COMPO_INFO NewTriStateCompoInfo;

    /* We sort the list of TriState 'LibcTriState' in order to          */
    /* prioritize the polarity matching between the pin 'Select'          */
    /* G_Current3StateComponent = TriState;                                */
    qsort (BListAdress (LibcTriState), BlistSize (LibcTriState),
           sizeof (LIBC_CELL), CmpLibcCellTriStateAreaPinMatch);

    if (BlistSize (LibcTriState) == 0)
    {
        fprintf (stderr,
" ERROR: Mapping aborted because no tri-states are available in the
library\n");
        exit (1);
    }

    Cell = (LIBC_CELL)BListElt (LibcTriState, 0);

    if (!LibCellWith3StatePin (Cell, &OutPin))
        return (GNL_MAP_ERROR);           /* should never happened ! */

    if (GnlCreateTriStateCompoInfo (&NewTriStateCompoInfo))
        return (GNL_MEMORY_FULL);

    SetGnlTriStateHook (TriState, NewTriStateCompoInfo);

    SetGnlTriStateCompoInfoCell (TriState, Cell);

    ListPinName = LibPinName (OutPin);
    if (!GnlSinglePinName (ListPinName))
        return (GNL_MAP_ERROR);           /* should never happened ! */
    SetGnlTriStateCompoInfoOutput (TriState,
                                   LibNameListName (ListPinName));

    Function = LibPinFunction (OutPin);
    ThreeState = LibPin3State (OutPin);

    SetGnlTriStateCompoInfoInput (TriState, Function);
    SetGnlTriStateCompoInfoEnable (TriState, ThreeState);

    if ((GnlStatus = GnlConstruct3StateLogicBoundaries (Gnl, TriState)))
        return (GnlStatus);

```

gnlmap3state.c

```
    return (GNL_OK);
}

/*-----*/
/* GnlMapList3States */
/*-----*/
GNL_STATUS GnlMapList3States (Gnl, Gnl3States, Libc3States)
    GNL          Gnl;
    BLIST        Gnl3States;
    BLIST        Libc3States;
{
    int           i;
    GNL_TRISTATE_COMPONENT   TriStatei;
    GNL_STATUS     GnlStatus;

    for (i=0; i<BListSize (Gnl3States); i++)
    {
        TriStatei = (GNL_TRISTATE_COMPONENT)BListElt (Gnl3States, i);
        if ((GnlStatus = GnlMap3STATE (Gnl, TriStatei, Libc3States)))
            return (GnlStatus);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlMap3State */
/*-----*/
/* This procedure maps the tri-state elements of the current 'Gnl'      */
/* on to the elemnts LIBC of the technology library. If the mapping of   */
/* one cell cannot be done then GNL_MAP_ERROR is returned, and GNL_OK if*/ 
/* everything is ok.                                                 */
/*-----*/
GNL_STATUS GnlMap3State (Gnl, GnlLibc)
    GNL          Gnl;
    LIBC_LIB GnlLibc;
{
    BLIST        Libc3States;
    BLIST        Gnl3States;
    GNL_STATUS   GnlStatus;
    GNL_LIB      HGnllib;

    HGnllib = (GNL_LIB)LibHook (GnlLibc);
    Libc3States = GnlHLibCells3States (HGnllib);

    if (GnlExtractGnl3States (Gnl, &Gnl3States))
        return (GNL_MEMORY_FULL);
    fprintf (stderr, "\n");

/*
    fprintf (stderr, "# LIBC: #3-STATES = %d\n", BListSize (Libc3States));
    fprintf (stderr, "# LIBC: #3-STATES = %d\n", BListSize (Gnl3States));

```

gnlmap3state.c

```
*/  
  
/* We map the Gnl 3-states with the LIBC 3-states. */  
if ((GnlStatus = GnlMapList3States (Gnl, Gnl3States, Libc3States)))  
    return (GnlStatus);  
  
return (GNL_OK);  
}  
  
/*----- EOF -----*/
```

gnlmapseq.c

```
/*-----*/
/*      File:          gnlmapseq.c
/*      Modifications: -
/*      Documentation: -
/*      Description: *
/*-----*/
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlmap.h"
#include "gnloption.h"

#include "blist.e"

/*-----*/
/* EXTERN
/*-----*/
extern GNL_ENV      G_GnlEnv;

/*-----*/
/* GnlCreateSequentialCompoInfo
/*-----*/
GNL_STATUS GnlCreateSequentialCompoInfo (NewSequentialCompoInfo)
    GNL_SEQUENTIAL_COMPO_INFO      *NewSequentialCompoInfo,
{
    if ((*NewSequentialCompoInfo = (GNL_SEQUENTIAL_COMPO_INFO)
        calloc (1, sizeof (GNL_SEQUENTIAL_COMPO_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlBoolOprIsSimpleSignal
/*-----*/
/* Returns 1 if the signal is simple and 0 if it is the NOT of a signal */
/* 'BoolOpr' is NULL. */
/*-----*/
int GnlBoolOprIsSimpleSignal (BoolOpr)
    LIBC_BOOL_OPR      BoolOpr;
{
```

gnlmapseq.c

```
if (!BoolOpr)
    return (0);

if (LibBoolOprType (BoolOpr) == ID_B)
    return (1);

return (0);
}

/*-----
/* GnlBoolOprIsSimpleFunction                                     */
/*-----*/
/* Either a simple signal or the NOT of a simple signal.        */
/*-----*/
int GnlBoolOprIsSimpleFunction (BoolOpr)
    LIBC_BOOL_OPR      BoolOpr;
{

    if (GnlBoolOprIsSimpleSignal (BoolOpr))
        return (1);

    if (LibBoolOprType (BoolOpr) != NOT_B)
        return (0);

    if (GnlBoolOprIsSimpleSignal (LibBoolOprRightSon (BoolOpr)))
        return (1);

    return (0);
}

/*-----
/* GnlPrintBoolOpr                                              */
/*-----*/
void GnlPrintBoolOpr (BoolOpr)
    LIBC_BOOL_OPR      BoolOpr;
{
    text_buffer          *VarName;

    if (!BoolOpr)
    {
        printf ("NULL");
        return;
    }

    /* terminal bool opr.                                         */
    switch (LibBoolOprType (BoolOpr)) {
        /* terminal case of a signal.                                */
        case ID_B:
            VarName = LibBoolOprIdName (BoolOpr);
            printf ("%s", VarName);
            break;

        case ZERO_B:
            printf ("1'b0");
            break;
    }
}
```

gnlmapseq.c

```
case ONE_B:
    printf ("1'b1");
    break;

/* case of binary Boolean operators */
case XOR_B:
    printf ("(");
    GnlPrintBoolOpr (LibBoolOprLeftSon (BoolOpr));
    printf (^(");
    GnlPrintBoolOpr (LibBoolOprRightSon (BoolOpr));
    printf (")");
    break;

case OR_B:
    printf ("(");
    GnlPrintBoolOpr (LibBoolOprLeftSon (BoolOpr));
    printf (^(");
    GnlPrintBoolOpr (LibBoolOprRightSon (BoolOpr));
    printf (")");
    break;

case AND_B:
    printf ("(");
    GnlPrintBoolOpr (LibBoolOprLeftSon (BoolOpr));
    printf (^(");
    GnlPrintBoolOpr (LibBoolOprRightSon (BoolOpr));
    printf (")");
    break;

case NOT_B:
    printf ("~(");
    GnlPrintBoolOpr (LibBoolOprRightSon (BoolOpr));
    printf (")");
    break;

default:
    fprintf (stderr,
             "Operator unsupported in 'GnlPrintBoolOpr'");
}
}

/*-----
/* GnlSeqCellCanBeSupported */
/*-----*/
/* This procedures verifies that the cell definition can be supported */
/* by the mapper and prints out a warning if it cannot. */
/*-----*/
int GnlSeqCellCanBeSupported (Cell)
    LIBC_CELL          Cell;
{
    LIBC_FF_LATCH  FFLatch;
    int             i;
    LIBC_PIN        Pins;
    LIBC_BOOL_OPR   PinFunction;
    LIBC_NAME_LIST  ListPinName;
```

gnlmapseq.c

```
Pins = LibCellPins (Cell);
for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    ListPinName = LibPinName (Pins);
    if (!GnlSinglePinName (ListPinName))
    {
        fprintf (stderr,
" WARNING: cell is ignored because of complex pin name: [%s]\n",
                LibCellName(Cell));
        return (0);
    }
    PinFunction = LibPinFunction (Pins);
    if (PinFunction)
    {
        if (!GnlBoolOprIsSimpleSignal (PinFunction))
        {
            fprintf (stderr,
" WARNING: cell is ignored because of complex output pin function: [%s]\n",
                LibCellName(Cell));
            return (0);
        }
    }
}

FFLatch = LibCellFFLatch (Cell);

if (LibFFLatchPreset (FFLatch) &&
    !GnlBoolOprIsSimpleFunction (LibFFLatchPreset (FFLatch)))
{
    fprintf (stderr,
" WARNING: cell is ignored because of complex Preset pin function: [%s]\n",
                LibCellName(Cell));
    return (0);
}

if (LibFFLatchClear (FFLatch) &&
    !GnlBoolOprIsSimpleFunction (LibFFLatchClear (FFLatch)))
{
    fprintf (stderr,
" WARNING: cell is ignored because of complex Clear pin function: [%s]\n",
                LibCellName(Cell));
    return (0);
}

if (LibFFLatchClockOn (FFLatch) &&
    !GnlBoolOprIsSimpleFunction (LibFFLatchClockOn (FFLatch)))
{
    fprintf (stderr,
" WARNING: cell is ignored because of complex Clock pin function: [%s]\n",
                LibCellName(Cell));
    return (0);
}

if (LibFFLatchEnable (FFLatch) &&
    !GnlBoolOprIsSimpleFunction (LibFFLatchEnable (FFLatch)))
```

gnlmapseq.c

```
{  
    fprintf (stderr,  
" WARNING: cell is ignored because of complex Enable pin function: [%s]\n",  
           LibCellName(Cell));  
    return (0);  
}  
  
if (LibFFLatchNextState (FFLatch) &&  
    !GnlBoolOprIsSimpleSignal (LibFFLatchNextState (FFLatch)))  
{  
    fprintf (stderr,  
" WARNING: cell is ignored because of complex next state function: [%s]\n",  
           LibCellName(Cell));  
    return (0);  
}  
  
if (LibFFLatchDataIn (FFLatch) &&  
    !GnlBoolOprIsSimpleSignal (LibFFLatchDataIn (FFLatch)))  
{  
    fprintf (stderr,  
" WARNING: cell is ignored because of complex data in function: [%s]\n",  
           LibCellName(Cell));  
    return (0);  
}  
  
if (LibFFLatchWidth (FFLatch) > 1)  
{  
    if (LibFFLatchIsFF (FFLatch))  
        fprintf (stderr,  
                " WARNING: bank of FFs of cell is ignored: [%s]\n",  
                LibCellName(Cell));  
    else  
        fprintf (stderr,  
                " WARNING: bank of Latches of cell is ignored: [%s]\n",  
                LibCellName(Cell));  
    return (0);  
}  
}  
  
/*-----*/  
/* GnlExtractLibcSequential */  
/*-----*/  
/* This procedure extracts from the technology library 'GnlLibc' the */  
/* lists of Flip-Flops and Latches and stores them in 'LibcDffs' and in */  
/* 'LibcLatches'. */  
/*-----*/  
GNL_STATUS GnlExtractLibcSequential (GnlLibc, LibcDffs, LibcLatches)  
{  
    LIBC_LIB      GnlLibc;  
    BLIST     *LibcDffs;  
    BLIST     *LibcLatches;  
    {  
        LIBC_CELL          CellI;  
        LIBC_FF_LATCH    FFLatch;  
  
        if (BListCreate (LibcDffs))
```

gnlmapseq.c

```
    return (GNL_MEMORY_FULL);

    if (BListCreate (LibcLatches))
        return (GNL_MEMORY_FULL);

    CellI = LibCells (GnlLibc);
    for ( ; CellI != NULL; CellI = LibCellNext (CellI))
    {
        /* If not a sequential cell or cell not to use */  

        if (!LibCellFFLatch (CellI) || LibCellDontUse (CellI))
            continue;

        /* If the cell can be supported by the mapping phase we add it */
        /* in the list. */  

        if (GnlSeqCellCanBeSupported (CellI))
        {
            FFLatch = LibCellFFLatch (CellI);
            if (LibFFLatchIsFF (FFLatch))
            {
                if (BListAddElt (*LibcDffs, (int)CellI))
                    return (GNL_MEMORY_FULL);
                }
            else
                {
                    if (BListAddElt (*LibcLatches, (int)CellI))
                        return (GNL_MEMORY_FULL);
                    }
            }
        }

    return (GNL_OK);
}

/*-----*/
/* GnlExtractGnlSequential */  

/*-----*/
GNL_STATUS GnlExtractGnlSequential (Gnl, GnlDffs, GnlLatches)
{
    GNL          Gnl;
    BLIST      *GnlDffs;
    BLIST      *GnlLatches;
    {  

        BLIST          Components;
        int           j;
        GNL_COMPONENT ComponentJ;
        GNL_SEQUENTIAL_COMPONENT SeqCompoJ;

        if (BListCreate (GnlDffs))
            return (GNL_MEMORY_FULL);

        if (BListCreate (GnlLatches))
            return (GNL_MEMORY_FULL);

        Components = GnlComponents (Gnl);
        if (!Components)
```

gnlmapseq.c

```
    return (GNL_OK);

for (j=0; j<BListSize (Components); j++)
{
    ComponentJ = (GNL_COMPONENT)BListElt (Components, j);
    if (GnlComponentType (ComponentJ) != GNL_SEQUENTIAL_COMPO)
        continue;

    SeqCompoJ = (GNL_SEQUENTIAL_COMPONENT)ComponentJ;

    /* If the component is a DFF flip-flop */ 
    if (GnlSeqComponentIsDFF (SeqCompoJ))
    {
        if (BListAddElt (*GnlDffs, (int)SeqCompoJ))
            return (GNL_MEMORY_FULL);
        continue;
    }

    /* If the component is a LATCH latch */ 
    if (GnlSeqComponentIsLATCH (SeqCompoJ))
    {
        if (BListAddElt (*GnlLatches, (int)SeqCompoJ))
            return (GNL_MEMORY_FULL);
        continue;
    }

    fprintf (stderr, " WARNING: unknown sequential element\n");
}

return (GNL_OK);
}

/*-----
/* GnlGetGnlNodeFromBoolOprWithSubstitute */
/*-----*/
/* This procedure returns the GNL_NODE expression thru 'GnlNode' of the */
/* original tree expression of type LIBC_BOOL_OPR generated in the LIBC */
/* parsing. Indexsignal, buses are not considered and we exit(1). */
/* The GNL_NODE objects are created in the memory segment of 'LibGnl'. */
/* Each leaf of BOOL_OPR is replaced by 'LeafGnlVar'. */
/*-----*/
GNL_STATUS GnlGetGnlNodeFromBoolOprWithSubstitute (LibGnl, LeafGnlVar,
                                                    BoolOpr, GnlNode)

    GNL           LibGnl;
    GNL_VAR       LeafGnlVar;
    LIBC_BOOL_OPR BoolOpr;
    GNL_NODE     *GnlNode;
{
    text_buffer   *VarName;
    GNL_STATUS   GnlStatus;
    GNL_VAR      GnlVar;
    GNL_NODE    GnlNodeLeft;
    GNL_NODE    GnlNodeRight;
    BLIST        NewList;

    /* terminal bool opr. */

```

```

switch (LibBoolOprType (BoolOpr)) {
    /* terminal case of a signal. */ */
    case ID_B:
        /* we do the substitution. */
        if (GnlCreateNodeForVar (LibGnl, LeafGnlVar, GnlNode))
            return (GNL_MEMORY_FULL);
        break;

    case ZERO_B:
        if (GnlCreateNodeVss (LibGnl, GnlNode))
            return (GNL_MEMORY_FULL);
        break;

    case ONE_B:
        if (GnlCreateNodeVdd (LibGnl, GnlNode))
            return (GNL_MEMORY_FULL);
        break;

    /* case of binary Boolean operators */
    case XOR_B:
    case OR_B:
    case AND_B:
        if ((GnlStatus = GnlGetGnlNodeFromBoolOprWithSubstitute (
                LibGnl,
                LeafGnlVar,
                LibBoolOprLeftSon (BoolOpr),
                &GnlNodeLeft)))
            return (GnlStatus);
        if ((GnlStatus = GnlGetGnlNodeFromBoolOprWithSubstitute (
                LibGnl,
                LeafGnlVar,
                LibBoolOprRightSon (BoolOpr),
                &GnlNodeRight)))
            return (GnlStatus);

        if (LibBoolOprType (BoolOpr) == XOR_B)
        {
            if (GnlCreateNodeXor (LibGnl, GnlNodeLeft, GnlNodeRight,
                    GnlNode, 0))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }

        if (BListCreateWithSize (2, &NewList))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (NewList, (int)GnlNodeLeft))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (NewList, (int)GnlNodeRight))
            return (GNL_MEMORY_FULL);

        if (GnlCreateNode (LibGnl,
                GnlOpFromBoolOprOp (LibBoolOprType (BoolOpr)),
                GnlNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*GnlNode, NewList);
        break;
}

```

gnlmapseq.c

```
/* Unary Boolean operators. Its son is the right one */  
case NOT_B:  
    if ((GnlStatus = GnlGetGnlNodeFromBoolOprWithSubstitute (LibGnl,  
                                                               LeafGnlVar,  
                                                               LibBoolOprRightSon (BoolOpr),  
                                                               &GnlNodeRight)))  
        return (GnlStatus);  
    if (GnlCreateNodeNot (LibGnl, GnlNodeRight, GnlNode))  
        return (GNL_MEMORY_FULL);  
    break;  
  
default:  
    fprintf (stderr,  
"Operator unsupported in 'GnlGetGnlNodeFromBoolOprWithSubstitute'");  
    return (GNL_MAP_ERROR);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlLookForSameExpression */  
/*-----*/  
int GnlLookForSameExpression (Gnl, Node, NewVar)  
    GNL          Gnl;  
    GNL_NODE     Node;  
    GNL_VAR      *NewVar;  
{  
    int           i;  
    GNL_VAR      Vari;  
    GNL_FUNCTION FunctionI;  
  
    for (i=BListSize (GnlFunctions (Gnl))-1; i>=0; i--)  
    {  
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  
        FunctionI = GnlVarFunction (Vari);  
  
        if (GnlEqualNode (Node, GnlFunctionOnSet (FunctionI)))  
        {  
            *NewVar = Vari;  
            return (1);  
        }  
    }  
    return (0);  
}  
  
/*-----*/  
/* GnlConstructDffLogicBoundaries */  
/*-----*/  
/* This procedure creates extra logic in the current Gnl by adding new */  
/* variables and functions in it. Functions are added if the polarities */  
/* are mis-matching between the LIB_CELL and its corresponding library */  
/* cell LIBC_CELL. Pins on which this can occur are the Set, Reset and */
```

gnlmapseq.c

```
/* clock signals. */  
/*-----*/  
GNL_STATUS GnlConstructDffLogicBoundaries (Gnl, Dff)  
    GNL          Gnl;  
    GNL_SEQUENTIAL_COMPONENT Dff;  
{  
    LIBC_BOOL_OPR Input;  
    char           *Output;  
    LIBC_BOOL_OPR Clock;  
    LIBC_BOOL_OPR Clear;  
    LIBC_BOOL_OPR Preset;  
    GNL_STATUS     GnlStatus;  
    GNL_VAR        ClearVar;  
    GNL_NODE       NewClearNode;  
    GNL_VAR        PresetVar;  
    GNL_NODE       NewPresetNode;  
    GNL_VAR        ClockVar;  
    GNL_NODE       NewClockNode;  
    GNL_VAR        NewVar;  
    GNL_FUNCTION   NewFunction;  
    GNL_NODE       NotNewClockNode;  
    GNL_VAR        VddVar;  
    GNL_NODE       VarNode;  
    GNL_VAR        OutputVar;  
    GNL_NODE       NewOutNode;  
  
    Input = GnlSeqCompoInfoInput (Dff);  
    Output = GnlSeqCompoInfoOutput (Dff);  
    Clock = GnlSeqCompoInfoClock (Dff);  
    Clear = GnlSeqCompoInfoReset (Dff);  
    Preset = GnlSeqCompoInfoSet (Dff);  
  
/*  
    GnlPrintBoolOpr (Input);  
    printf ("\n");  
    printf ("%s", Output);  
    printf ("\n");  
    GnlPrintBoolOpr (Clock);  
    printf ("\n");  
    GnlPrintBoolOpr (Clear);  
    printf ("\n");  
    GnlPrintBoolOpr (Preset);  
    printf ("\n");  
*/  
  
/* The sequential element selected in the library has only a QBAR */  
/* pin so we need to create an inverter in front. */  
if (GnlSequentialCompoFormOutput (Dff) == GNL_QBAR)  
{  
    OutputVar = GnlSequentialCompoOutput (Dff);  
    if (GnlCreateUniqueVar (Gnl, GnlVarName (OutputVar), &NewVar))  
        return (GNL_MEMORY_FULL);  
}
```

```

/* The output var becomes to be a local. */  

if (!BListMemberOfList (GnlLocals (Gnl), OutputVar, IntIdentical))  

{  

    if (BListAddElt (GnlLocals (Gnl), (int) NewVar))  

        return (GNL_MEMORY_FULL);  

    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);  

}  

SetGnlSequentialCompoOutput (Dff, NULL);  

SetGnlSequentialCompoOutputBar (Dff, NewVar);  

SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);  

/* Creating the logic which is an inverter */  

if (GnlCreateNodeForVar (Gnl, NewVar, &VarNode))  

    return (GNL_MEMORY_FULL);  

if (GnlCreateNodeNot (Gnl, VarNode, &NewOutNode))  

    return (GNL_MEMORY_FULL);  

if (GnlVarFunction (OutputVar))  

{  

    fprintf (stderr,  

" ERROR: multi-source on signal <%s> during Flip-Flop logic fracturing.\n",
           GnlVarName (OutputVar));  

    exit (1);
}  

if (GnlFunctionCreate (Gnl, OutputVar, NULL, &NewFunction))  

    return (GNL_MEMORY_FULL);  

SetGnlVarFunction (OutputVar, NewFunction);  

SetGnlFunctionVar (NewFunction, OutputVar);  

SetGnlFunctionOnSet (NewFunction, NewOutNode);  

if (BListAddElt (GnlFunctions (Gnl), (int) OutputVar))  

    return (GNL_MEMORY_FULL);  

}  

/* We need to create extra logic for the 'Clear' */  

/* because clear of LIBC_CELL and LIB_CELL have opposite polarities */  

if (Clear &&  

    ((!GnlBoolOprIsSimpleSignal (Clear) &&  

     (GnlSequentialCompoResetPol (Dff) == 1)) ||  

     (GnlBoolOprIsSimpleSignal (Clear) &&  

     (GnlSequentialCompoResetPol (Dff) == 0))))  

{  

    ClearVar = GnlSequentialCompoReset (Dff);  

    /* Creating the logic which is an inverter */  

    if (GnlCreateNodeForVar (Gnl, ClearVar, &VarNode))  

        return (GNL_MEMORY_FULL);  

    if (GnlCreateNodeNot (Gnl, VarNode, &NewClearNode))  

        return (GNL_MEMORY_FULL);  

    /* Look if this expression already exists ... */  

    if (GnlLookForSameExpression (Gnl, NewClearNode, &NewVar))
}

```

gnlmapseq.c

```
{  
    SetGnlSequentialCompoReset (Dff, NewVar);  
    if (!GnlVarIsPrimary (NewVar))  
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);  
}  
else  
{  
    if (GnlCreateUniqueVar (Gnl, GnlVarName (ClearVar), &NewVar))  
        return (GNL_MEMORY_FULL);  
    if (BListAddElt (GnlLocals (Gnl), (int)NewVar))  
        return (GNL_MEMORY_FULL);  
    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);  
  
    if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))  
        return (GNL_MEMORY_FULL);  
  
    SetGnlVarFunction (NewVar, NewFunction);  
    SetGnlFunctionVar (NewFunction, NewVar);  
    SetGnlFunctionOnSet (NewFunction, NewClearNode);  
  
    if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))  
        return (GNL_MEMORY_FULL);  
  
    SetGnlSequentialCompoReset (Dff, NewVar);  
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);  
}  
}  
  
/* We need to create extra logic for the 'Preset' */  
/* because Set of LIBC_CELL and LIB_CELL have opposite polarities */  
if (Preset &&  
    ((!GnlBoolOprIsSimpleSignal (Preset) &&  
     (GnlSequentialCompoSetPol (Dff) == 1)) ||  
     (GnlBoolOprIsSimpleSignal (Preset) &&  
     (GnlSequentialCompoSetPol (Dff) == 0))))  
{  
    PresetVar = GnlSequentialCompoSet (Dff);  
  
    /* Creating the logic which is an inverter */  
    if (GnlCreateNodeForVar (Gnl, PresetVar, &VarNode))  
        return (GNL_MEMORY_FULL);  
    if (GnlCreateNodeNot (Gnl, VarNode, &NewPresetNode))  
        return (GNL_MEMORY_FULL);  
  
    /* Look if this expression already exists ... */  
    if (GnlLookForSameExpression (Gnl, NewPresetNode, &NewVar))  
    {  
        SetGnlSequentialCompoSet (Dff, NewVar);  
        if (!GnlVarIsPrimary (NewVar))  
            SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);  
    }  
    else  
    {  
        if (GnlCreateUniqueVar (Gnl, GnlVarName (PresetVar), &NewVar))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (GnlLocals (Gnl), (int)NewVar))  
            return (GNL_MEMORY_FULL);  
    }
```

gnlmapseq.c

```
SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (NewVar, NewFunction);
SetGnlFunctionVar (NewFunction, NewVar);
SetGnlFunctionOnSet (NewFunction, NewPresetNode);

if (BListAddElt (GnlFunctions(Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);

SetGnlSequentialCompoSet (Dff, NewVar);
SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
}

}

/* We need to create extra logic for the 'Clock'. */
/* This is the case if the LIB_CELL and LIBC_CELL have different */
/* clock polarities. */
if ((!GnlBoolOprIsSimpleSignal (Clock) &&
     (GnlSequentialCompoClockPol (Dff) == 1)) ||
    (GnlBoolOprIsSimpleSignal (Clock) &&
     (GnlSequentialCompoClockPol (Dff) == 0)))
{
    ClockVar = GnlSequentialCompoClock (Dff);

    fprintf (stderr,
             " WARNING: creating gated clock for Flip-Flop <%s>\n",
             GnlSequentialCompoInstName (Dff));
    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, ClockVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewClockNode))
        return (GNL_MEMORY_FULL);

    /* Look if this expression already exists ... */
    if (GnlLookForSameExpression (Gnl, NewClockNode, &NewVar))
    {
        SetGnlSequentialCompoClock (Dff, NewVar);
        if (!GnlVarIsPrimary (NewVar))
            SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
    else
    {
        if (GnlCreateUniqueVar (Gnl, GnlVarName (ClockVar), &NewVar))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals(Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

        if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);

        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionVar (NewFunction, NewVar);
        SetGnlFunctionOnSet (NewFunction, NewClockNode);
```

gnlmapseq.c

```
if (BListAddElt (GnlFunctions (Gnl), (int) NewVar))
    return (GNL_MEMORY_FULL);

    SetGnlSequentialCompoClock (Dff, NewVar);
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
}
}

return (GNL_OK);
}

/*-----*/
/* GnlUpdateSequentialFormOutput */
/*-----*/
/* We analyze the accociated tech. cell 'FFLatch' to see if it has */
/* a Q and QBAR, or only Q, or only QBAR outputs. */
/*-----*/
void GnlUpdateSequentialFormOutput (SeqCompo, Cell)
    GNL_SEQUENTIAL_COMPONENT      SeqCompo;
    LIBC_CELL                      Cell;
{
    int                         OutForm;
    LIBC_PIN                     Pins;
    LIBC_BOOL_OPR                PinFunction;
    char                         *IQ;
    char                         *IQN;

    OutForm = 0;

    IQ = LibFFLatchQName (LibCellFFLatch (Cell));
    IQN = LibFFLatchQName (LibCellFFLatch (Cell));

    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext (Pins))
    {
        PinFunction = LibPinFunction (Pins);
        if (PinFunction)
        {
            if (GnlNameIsPresent (IQN, PinFunction))
                OutForm += 2;
            else if (GnlNameIsPresent (IQ, PinFunction))
                OutForm += 1;
        }
    }

    switch (OutForm) {
        case 1:
            SetGnlSequentialCompoFormOutput (SeqCompo, GNL_Q);
            break;

        case 2:
            SetGnlSequentialCompoFormOutput (SeqCompo, GNL_QBAR);
            break;
    }
}
```

gnlmapseq.c

```
case 3:
    SetGnlSequentialCompoFormOutput (SeqCompo, GNL_Q_QBAR);
    break;

default:
    fprintf (stderr,
" ERROR: Mapit cannot map on sequential element <%s> because it has no IQ and
IQN pins declared\n",
        LibCellName (Cell));
    exit (1);
}

/*
/* GnlMatchDFF
*/
/* This procedure verifies if LIBC cell 'Cell' can realize the Gnl flip */
/* flop 'Dff'. We consider only the Q outputs (and not QN output) of the*/
/* cell 'Cell'.
*/
GNL_STATUS GnlMatchDFF (Gnl, Dff, Cell, Match)
    GNL                      Gnl;
    GNL_SEQUENTIAL_COMPONENT   Dff;
    LIBC_CELL                  Cell;
    int                        *Match;
{
    LIBC_FF_LATCH              FFLatch;
    GNL_SEQUENTIAL_COMPO_INFO NewSequentialCompoInfo;
    GNL_STATUS                 GnlStatus;

    *Match = 0;
    FFLatch = LibCellFFLatch (Cell);

    /* If FF with reset ... */
    if (GnlSequentialCompoReset (Dff) &&
        !GnlVarIsVss (GnlSequentialCompoReset (Dff)))
        /* If the Cell has no Clear then it is not powerfull enough. */
        if (!LibFFLatchClear (FFLatch))
            return (GNL_OK);

    /* If FF with Set ... */
    if (GnlSequentialCompoSet (Dff) &&
        !GnlVarIsVss (GnlSequentialCompoSet (Dff)))
        /* If the Cell has no Preset then it is not powerfull enough. */
        if (!LibFFLatchPreset (FFLatch))
            return (GNL_OK);

    /* No clock definition ... cannot be used for a Dff. */
    if (!LibFFLatchClockOn (FFLatch))
        return (GNL_OK);

    /* If we have both reset and set we verify that the FF value when
     * both are active corresponds to the one in 'LibFFLatchClPrVar1'
     */
    if (GnlSequentialCompoReset (Dff) &&
        !GnlVarIsVss (GnlSequentialCompoReset (Dff)) &&
        GnlSequentialCompoSet (Dff) &&
```

gnlmapseq.c

```
!GnlVarIsVss (GnlSequentialCompoSet (Dff)))
{
    switch (GnlSequentialCompoOp (Dff)) {
        case GNL_DFF:
            fprintf (stderr,
" WARNING: use of a 'dff' component using set and reset pins\n");
            fprintf (stderr,
" consider it as a 'dffx' component\n");

        case GNL_DFFX:
            /* we expect everything except 'L' and 'H' as value */
            if ((LibFFLatchClPrVar1 (FFLatch) == 'L') ||
                (LibFFLatchClPrVar1 (FFLatch) == 'H'))
                return (GNL_OK);
            break;

        case GNL_DFF0:
            /* we expect 'L'. */
            if (LibFFLatchClPrVar1 (FFLatch) != 'L')
                return (GNL_OK);
            break;

        case GNL_DFF1:
            /* we expect 'H'. */
            if (LibFFLatchClPrVar1 (FFLatch) != 'H')
                return (GNL_OK);
            break;
    }

/* so we found a sequential element on which we can realize the Dff */
if (GnlCreateSequentialCompoInfo (&NewSequentialCompoInfo))
    return (GNL_MEMORY_FULL);
SetGnlSequentialCompoHook (Dff, NewSequentialCompoInfo);

SetGnlSeqCompoInfoCell (Dff, Cell);
SetGnlSeqCompoInfoInput (Dff, LibFFLatchNextState (FFLatch));
SetGnlSeqCompoInfoOutput (Dff, LibFFLatchQName (FFLatch));
SetGnlSeqCompoInfoClock (Dff, LibFFLatchClockOn (FFLatch));

/* If FF with reset ... */
if (GnlSequentialCompoReset (Dff) &&
    !GnlVarIsVss (GnlSequentialCompoReset (Dff)))
{
    SetGnlSeqCompoInfoReset (Dff, LibFFLatchClear (FFLatch));
}

/* If FF with Set ... */
if (GnlSequentialCompoSet (Dff) &&
    !GnlVarIsVss (GnlSequentialCompoSet (Dff)))
{
    SetGnlSeqCompoInfoSet (Dff, LibFFLatchPreset (FFLatch));
}

/* We analyze the accociated tech. cell 'FFLatch' to see if it has */
/* a Q and QBAR, or only Q, or only QBAR outputs. */
GnlUpdateSequentialFormOutput (Dff, Cell);
```

gnlmapseq.c

```
if ((GnlStatus = GnlConstructDffLogicBoundaries (Gnl, Dff)))  
    return (GnlStatus);  
  
*Match = 1;  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlConstructLatchLogicBoundaries */  
/*-----*/  
/* This procedure creates extra logic in the current Gnl by adding new */  
/* variables and functions in it. Functions are added if the polarities */  
/* are mis-matching between the LIB_CELL and its corresponding library */  
/* cell LIBC_CELL. Pins on which this can occur are the Set, Reset and */  
/* clock signals. */  
/*-----*/  
GNL_STATUS GnlConstructLatchLogicBoundaries (Gnl, Latch)  
    GNL                      Gnl;  
    GNL_SEQUENTIAL_COMPONENT  Latch;  
{  
    LIBC_BOOL_OPR           Input;  
    char                    *Output;  
    LIBC_BOOL_OPR           Clock;  
    LIBC_BOOL_OPR           Clear;  
    LIBC_BOOL_OPR           Preset;  
    GNL_STATUS              GnlStatus;  
    GNL_VAR                 ClearVar;  
    GNL_NODE                NewClearNode;  
    GNL_VAR                 PresetVar;  
    GNL_NODE                NewPresetNode;  
    GNL_VAR                 ClockVar;  
    GNL_NODE                NewClockNode;  
    GNL_VAR                 NewVar;  
    GNL_FUNCTION             NewFunction;  
    GNL_NODE                NotNewClockNode;  
    GNL_VAR                 VddVar;  
    GNL_NODE                VarNode;  
    GNL_VAR                 OutputVar;  
    GNL_NODE                NewOutNode;  
  
    Input = GnlSeqCompoInfoInput (Latch);  
    Output = GnlSeqCompoInfoOutput (Latch);  
    Clock = GnlSeqCompoInfoClock (Latch);  
    Clear = GnlSeqCompoInfoReset (Latch);  
    Preset = GnlSeqCompoInfoSet (Latch);  
  
/*  
    GnlPrintBoolOpr (Input);  
    printf ("\n");  
    printf ("%s", Output);  
    printf ("\n");
```

gnlmapseq.c

```
GnlPrintBoolOpr (Clock);
printf ("\n");
GnlPrintBoolOpr (Clear);
printf ("\n");
GnlPrintBoolOpr (Preset);
printf ("\n");
*/
/* The sequential element selected in the library has only a QBAR */
/* pin so we need to create an inverter in front. */
if (GnlSequentialCompoFormOutput (Latch) == GNL_QBAR)
{
    OutputVar = GnlSequentialCompoOutput (Latch);
    if (GnlCreateUniqueVar (Gnl, GnlVarName (OutputVar), &NewVar))
        return (GNL_MEMORY_FULL);

    /* The output var becomes to be a local. */
    if (!BListMemberOfList (GnlLocals (Gnl), OutputVar, IntIdentical))
    {
        if (BListAddElt (GnlLocals (Gnl), (int) NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
    }

    SetGnlSequentialCompoOutput (Latch, NULL);

    SetGnlSequentialCompoOutputBar (Latch, NewVar);
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);

    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, NewVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewOutNode))
        return (GNL_MEMORY_FULL);

    if (GnlVarFunction (OutputVar))
    {
        fprintf (stderr,
" ERROR: multi-source on signal <%s> during Latch logic fracturing.\n",
                GnlVarName (OutputVar));
        exit (1);
    }

    if (GnlFunctionCreate (Gnl, OutputVar, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (OutputVar, NewFunction);
    SetGnlFunctionVar (NewFunction, OutputVar);
    SetGnlFunctionOnSet (NewFunction, NewOutNode);

    if (BListAddElt (GnlFunctions (Gnl), (int) OutputVar))
        return (GNL_MEMORY_FULL);
}

/* We need to create extra logic for the 'Clear'
/* because clear of LIBC_CELL and LIB_CELL have opposite polarities */

```

gnlmapseq.c

```
if (Clear &&
    ((!GnlBoolOprIsSimpleSignal (Clear) &&
      (GnlSequentialCompoResetPol (Latch) == 1)) ||
     (GnlBoolOprIsSimpleSignal (Clear) &&
      (GnlSequentialCompoResetPol (Latch) == 0)))) {
    ClearVar = GnlSequentialCompoReset (Latch);

    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, ClearVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewClearNode))
        return (GNL_MEMORY_FULL);

    /* Look if this expression already exists ... */
    if (GnlLookForSameExpression (Gnl, NewClearNode, &NewVar))
    {
        SetGnlSequentialCompoReset (Latch, NewVar);
        if (!GnlVarIsPrimary (NewVar))
            SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
    else
    {
        if (GnlCreateUniqueVar (Gnl, GnlVarName (ClearVar), &NewVar))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

        if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);

        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionVar (NewFunction, NewVar);
        SetGnlFunctionOnSet (NewFunction, NewClearNode);

        if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);

        SetGnlSequentialCompoReset (Latch, NewVar);
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
}

/* We need to create extra logic for the 'Preset' */
/* because Set of LIBC_CELL and LIB_CELL have opposite polarities */
if (Preset &&
    ((!GnlBoolOprIsSimpleSignal (Preset) &&
      (GnlSequentialCompoSetPol (Latch) == 1)) ||
     (GnlBoolOprIsSimpleSignal (Preset) &&
      (GnlSequentialCompoSetPol (Latch) == 0)))) {
    PresetVar = GnlSequentialCompoSet (Latch);

    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, PresetVar, &VarNode))
        return (GNL_MEMORY_FULL);
```

gnlmapseq.c

```
if (GnlCreateNodeNot (Gnl, VarNode, &NewPresetNode))
    return (GNL_MEMORY_FULL);

/* Look if this expression already exists ... */
if (GnlLookForSameExpression (Gnl, NewPresetNode, &NewVar))
{
    SetGnlSequentialCompoSet (Latch, NewVar);
    if (!GnlVarIsPrimary (NewVar))
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
}
else
{
    if (GnlCreateUniqueVar (Gnl, GnlVarName (PresetVar), &NewVar))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

    if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (NewVar, NewFunction);
    SetGnlFunctionVar (NewFunction, NewVar);
    SetGnlFunctionOnSet (NewFunction, NewPresetNode);

    if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
        return (GNL_MEMORY_FULL);

    SetGnlSequentialCompoSet (Latch, NewVar);
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
}
}

/* We need to create extra logic for the 'Clock'. */
/* This is the case if the LIB_CELL and LIBC_CELL have different */
/* clock polarities. */
if ((!GnlBoolOprIsSimpleSignal (Clock) &&
     (GnlSequentialCompoClockPol (Latch) == 1)) ||
    (GnlBoolOprIsSimpleSignal (Clock) &&
     (GnlSequentialCompoClockPol (Latch) == 0)))
{
    ClockVar = GnlSequentialCompoClock (Latch);

    fprintf (stderr,
             " WARNING: creating gated clock for Latch <%s>\n",
             GnlSequentialCompoInstName (Latch));
    /* Creating the logic which is an inverter */
    if (GnlCreateNodeForVar (Gnl, ClockVar, &VarNode))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, VarNode, &NewClockNode))
        return (GNL_MEMORY_FULL);

    /* Look if this expression already exists ... */
    if (GnlLookForSameExpression (Gnl, NewClockNode, &NewVar))
    {
        SetGnlSequentialCompoClock (Latch, NewVar);
        if (!GnlVarIsPrimary (NewVar))
```

gnlmapseq.c

```
        SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
    }
else
{
    if (GnlCreateUniqueVar (Gnl, GnlVarName (ClockVar), &NewVar))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (GnlLocals(Gnl), (int)NewVar))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

    if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (NewVar, NewFunction);
    SetGnlFunctionVar (NewFunction, NewVar);
    SetGnlFunctionOnSet (NewFunction, NewClockNode);

    if (BListAddElt (GnlFunctions(Gnl), (int)NewVar))
        return (GNL_MEMORY_FULL);

    SetGnlSequentialCompoClock (Latch, NewVar);
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);
}
}

return (GNL_OK);
}

/*
 * GnlMatchLATCH
 */
/*
 * This procedure verifies if LIBC cell 'Cell' can realize the Gnl Latch*
 * 'Latch'. We consider only the Q outputs (and not QN output) of the   *
 * cell 'Cell'.
 */
GNL_STATUS GnlMatchLATCH (Gnl, Latch, Cell, Match)
    GNL                      Gnl;
    GNL_SEQUENTIAL_COMPONENT  Latch;
    LIBC_CELL                 Cell;
    int                       *Match;
{
    LIBC_FF_LATCH             FFLatch;
    GNL_SEQUENTIAL_COMPO_INFO NewSequentialCompoInfo;
    GNL_STATUS                GnlStatus;

    *Match = 0;
    FFLatch = LibCellFFLatch (Cell);

    /* If Latch with reset ... */
    if (GnlSequentialCompoReset (Latch) &&
        !GnlVarIsVss (GnlSequentialCompoReset (Latch)))
    /* If the Cell has no Clear then it is not powerfull enough. */
    if (!LibFFLatchClear (FFLatch))
        return (GNL_OK);
}
```

```

/* If Latch with Set ... */  

if (GnlSequentialCompoSet (Latch) &&  

    !GnlVarIsVss (GnlSequentialCompoSet (Latch)))  

/* If the Cell has no Preset then it is not powerfull enough. */  

if (!LibFFLatchPreset (FFLatch))  

    return (GNL_OK);  
  

/* If we have both reset and set we verify that the LATCH value when */  

/* both are active corresponds to the one in 'LibFFLatchClPrVar1' */  

if (GnlSequentialCompoReset (Latch) &&  

    !GnlVarIsVss (GnlSequentialCompoReset (Latch)) &&  

    GnlSequentialCompoSet (Latch) &&  

    !GnlVarIsVss (GnlSequentialCompoSet (Latch)))  

{  

    switch (GnlSequentialCompoOp (Latch)) {  

        case GNL_LATCH:  

            fprintf (stderr,  

                " WARNING: use of a 'latch' component using set and reset pins\n");  

            fprintf (stderr,  

                " consider it as a 'latchx' component\n");  
  

        case GNL_LATCHX:  

            /* we expect everything except '0' and '1' as value */  

            if ((LibFFLatchClPrVar1 (FFLatch) == 'L') ||  

                (LibFFLatchClPrVar1 (FFLatch) == 'H'))  

                return (GNL_OK);  

            break;  
  

        case GNL_LATCH0:  

            /* we expect 'L'. */  

            if (LibFFLatchClPrVar1 (FFLatch) != 'L')  

                return (GNL_OK);  

            break;  
  

        case GNL_LATCH1:  

            /* we expect 'H'. */  

            if (LibFFLatchClPrVar1 (FFLatch) != 'H')  

                return (GNL_OK);  

            break;  

    }  

}  
  

/* so we found a sequential element on which we can realize the Latch*/  

if (GnlCreateSequentialCompoInfo (&NewSequentialCompoInfo))  

    return (GNL_MEMORY_FULL);  

SetGnlSequentialCompoHook (Latch, NewSequentialCompoInfo);  
  

SetGnlSeqCompoInfoCell (Latch, Cell);  

SetGnlSeqCompoInfoInput (Latch, LibFFLatchDataIn (FFLatch));  

SetGnlSeqCompoInfoOutput (Latch, LibFFLatchQName (FFLatch));  
  

/* The clock ofthe latch is actually the enable pin of the LIBC cell */  

SetGnlSeqCompoInfoClock (Latch, LibFFLatchEnable (FFLatch));  
  

/* If Latch with reset ... */  

if (GnlSequentialCompoReset (Latch) &&

```

gnlmapseq.c

```
!GnlVarIsVss (GnlSequentialCompoReset (Latch))
{
    SetGnlSeqCompoInfoReset (Latch, LibFFLatchClear (FFLatch));
}

/* If Latch with Set ... */
if (GnlSequentialCompoSet (Latch) &&
    !GnlVarIsVss (GnlSequentialCompoSet (Latch)))
{
    SetGnlSeqCompoInfoSet (Latch, LibFFLatchPreset (FFLatch));
}

/* We analyze the accociated tech. cell 'FFLatch' to see if it has */
/* a Q and QBAR, or only Q, or only QBAR outputs. */
GnlUpdateSequentialFormOutput (Latch, Cell);

if ((GnlStatus = GnlConstructLatchLogicBoundaries (Gnl, Latch)))
    return (GnlStatus);

*Match = 1;

return (GNL_OK);
}

/*
 * CmpLibcCellDffAreaPinMatch
 */
/*
 * Compare 'cell1' vs. 'cell2' by taking into account feature of the
 * current Sequential component 'G_CurrentSeqComponent'. The ordering
 * is done in priority to take care about the polarity of the pins of
 * 'G_CurrentSeqComponent' which are in the order: Clock, Set and Reset.
 * The LIBC_CELL matching the most the polarities is prioritized.
 */
static GNL_SEQUENTIAL_COMPONENT          G_CurrentSeqComponent;

static int CmpLibcCellDffAreaPinMatch (Cell1, Cell2)
    LIBC_CELL      *Cell1;
    LIBC_CELL      *Cell2;
{
    LIBC_FF_LATCH           FFLatch1;
    LIBC_FF_LATCH           FFLatch2;

    FFLatch1 = LibCellFFLatch (*Cell1);
    FFLatch2 = LibCellFFLatch (*Cell2);

    /* Rising clock. */
    if (GnlSequentialCompoClockPol (G_CurrentSeqComponent) == 1)
    {
        if (GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch1)) &&
            !GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch2)) &&
            GnlEnvPinSeqPolMatching ())
            return (-1);
        if (!GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch1)) &&
            GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch2)) &&
            GnlEnvPinSeqPolMatching ())
            return (1);
    }
}
```

gnlmapseq.c

```
        }
    else
        /* Falling clock                                     */
    {
        if (GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch1)) &&
            !GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch2)) &&
            GnlEnvPinSeqPolMatching ())
            return (1);
        if (!GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch1)) &&
            GnlBoolOprIsSimpleSignal (LibFFLatchClockOn (FFLatch2)) &&
            GnlEnvPinSeqPolMatching ())
            return (-1);
    }

    /* Everything is ok regarding the polarity of the pins so we decide   */
    /* with the area.                                                       */
    /* We privilegeate the minimum area                                     */
    if (LibCellArea(*Cell1) < LibCellArea(*Cell2))
        return (-1);

    if (LibCellArea(*Cell1) == LibCellArea(*Cell2))
        return (0);

    return (1);
}

/*-----*/
/* CmpLibcCellLatchAreaPinMatch                                     */
/*-----*/
/* Comapre 'cell1' vs. 'cell2' by taking into account feature of the */
/* current Sequential component 'G_CurrentSeqComponent'. The ordering */
/* is done in priority to take care about the polarity of the pins of */
/* 'G_CurrentSeqComponent' which are in the order: Clock, Set and Reset.*/
/* The LIBC_CELL matching the most the polarities is prioritized.      */
/*-----*/
static int CmpLibcCellLatchAreaPinMatch (Cell1, Cell2)
    LIBC_CELL      *Cell1;
    LIBC_CELL      *Cell2;
{
    LIBC_FF_LATCH          FFLatch1;
    LIBC_FF_LATCH          FFLatch2;

    FFLatch1 = LibCellFFLatch (*Cell1);
    FFLatch2 = LibCellFFLatch (*Cell2);

    /* High level clock.                                              */
    if (GnlSequentialCompoClockPol (G_CurrentSeqComponent) == 1)
    {
        if (GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch1)) &&
            !GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch2)) &&
            GnlEnvPinSeqPolMatching ())
            return (-1);
        if (!GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch1)) &&
            GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch2)) &&
```

gnlmapseq.c

```
        GnlEnvPinSeqPolMatching ())
    return (1);
}
else
/* Low level clock */
{
    if (GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch1)) &&
        !GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch2)) &&
        GnlEnvPinSeqPolMatching ())
    return (1);
    if (!GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch1)) &&
        GnlBoolOprIsSimpleSignal (LibFFLatchEnable (FFLatch2)) &&
        GnlEnvPinSeqPolMatching ())
    return (-1);
}

/* Everything is ok regarding the polarity of the pins so we decide */
/* with the area. */
/* We privilegate the minimum area */
if (LibCellArea(*Cell1) < LibCellArea(*Cell2))
    return (-1);

if (LibCellArea(*Cell1) == LibCellArea(*Cell2))
    return (0);

return (1);
}

-----*/
/* GnlMapDFF */
-----*/
GNL_STATUS GnlMapDFF (Gnl, Dff, LibcDffs)
    GNL                      Gnl;
    GNL_SEQUENTIAL_COMPONENT   Dff;
    BLIST                     LibcDffs;
{
    int                         i;
    LIBC_CELL                   CellI;
    int                         Match;
    GNL_STATUS                  GnlStatus;

/* We sort the list of Dff 'LibcDffs' in order to prioritize the
/* polarity matching between the pins Clock, Reset, Set */
G_CurrentSeqComponent = Dff;
qsort (BListAdress (LibcDffs), BListSize (LibcDffs),
       sizeof (LIBC_CELL), CmpLibcCellDffAreaPinMatch);

for (i=0; i<BListSize (LibcDffs); i++)
{
    CellI = (LIBC_CELL)BListElt (LibcDffs, i);
    if ((GnlStatus = GnlMatchDFF (Gnl, Dff, CellI, &Match)))
        return (GNL_MEMORY_FULL);

/* If the Match was successful we return. */
}
```

gnlmapseq.c

```
if (Match)
    return (GNL_OK);
}

fprintf (stderr, " MAP-ERROR: cannot map instance dff <%s>\n",
        GnlSequentialCompoInstName (Dff));

return (GNL_MAP_ERROR);
}

/*-----
/* GnlMapLATCH
*/
-----*/
GNL_STATUS GnlMapLATCH (Gnl, Latch, LibcLatches)
    GNL          Gnl;
    GNL_SEQUENTIAL_COMPONENT   Latch;
    BLIST         LibcLatches;
{
    int           i;
    LIBC_CELL     CellI;
    int           Match;
    GNL_STATUS    GnlStatus;

/* We sort the list of Dff 'LibcLatches' in order to prioritize the */
/* polarity matching between the pins Clock, Reset, Set           */
G_CurrentSeqComponent = Latch;
qsort (BListAddress (LibcLatches), BListSize (LibcLatches),
       sizeof (LIBC_CELL), CmpLibcCellLatchAreaPinMatch);

for (i=0; i<BListSize (LibcLatches); i++)
{
    CellI = (LIBC_CELL)BListElt (LibcLatches, i);
    if ((GnlStatus = GnlMatchLATCH (Gnl, Latch, CellI, &Match)))
        return (GNL_MEMORY_FULL);

    /* If the Match was successful we return.                      */
    if (Match)
        return (GNL_OK);
}

fprintf (stderr, " MAP-ERROR: cannot map instance latch <%s>\n",
        GnlSequentialCompoInstName (Latch));

return (GNL_MAP_ERROR);
}

/*-----
/* GnlMapListDFF
*/
-----*/
GNL_STATUS GnlMapListDFF (Gnl, GnlDffs, LibcDffs)
    GNL          Gnl;
    BLIST        GnlDffs;
    BLIST        LibcDffs;
{
    int           i;
    GNL_SEQUENTIAL_COMPONENT   DFFi;
```

gnlmapseq.c

```
GNL_STATUS GnlStatus;

for (i=0; i<BListSize (GnlDffs); i++)
{
    DFFi = (GNL_SEQUENTIAL_COMPONENT)BListElt (GnlDffs, i);
    if ((GnlStatus = GnlMapDFF (Gnl, DFFi, LibcDffs)))
        return (GnlStatus);
}

return (GNL_OK);
}

/*-----*/
/* GnlMapListLATCH */
/*-----*/
GNL_STATUS GnlMapListLATCH (Gnl, GnlLatches, LibcLatches)
    GNL          Gnl;
    BLIST        GnlLatches;
    BLIST        LibcLatches;
{
    int           i;
    GNL_SEQUENTIAL_COMPONENT LATCHi;
    GNL_STATUS    GnlStatus;

    for (i=0; i<BListSize (GnlLatches); i++)
    {
        LATCHi = (GNL_SEQUENTIAL_COMPONENT)BListElt (GnlLatches, i);
        if ((GnlStatus = GnlMapLATCH (Gnl, LATCHi, LibcLatches)))
            return (GnlStatus);
    }

    return (GNL_OK);
}

/*-----*/
/* CmpLibcCellArea */
/*-----*/
/* Returns the comparison between the area of two cells */
/*-----*/
static int CmpLibcCellArea (Cell1, Cell2)
    LIBC_CELL    *Cell1;
    LIBC_CELL    *Cell2;
{
    if (LibCellArea(*Cell1) < LibCellArea(*Cell2))
        return (-1);

    if (LibCellArea(*Cell1) == LibCellArea(*Cell2))
        return (0);

    return (1);
}

/*-----*/
```

gnlmapseq.c

```
/* GnlCreateOutputVarReplaceInNode */  
/*-----*/  
/* This procedure replace Node which are Not function of an output */  
/* sequential element and connect to the Qbar of the seq. element */  
/*-----*/  
char G_VarBarName[528];  
GNL_STATUS GnlCreateOutputVarReplaceInNode (Gnl, Node, NewNode)  
    GNL           Gnl;  
    GNL_NODE     Node;  
    GNL_NODE *NewNode;  
{  
    GNL_NODE      Son;  
    GNL_VAR       Var;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    GNL_VAR       OutputVarBar;  
    int          i;  
    GNL_NODE     NewSon;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
    {  
        *NewNode = Node;  
        return (GNL_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        *NewNode = Node;  
        return (GNL_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_NOT)  
    {  
        Son = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);  
        if (GnlNodeOp (Son) == GNL_VARIABLE)  
        {  
            Var = (GNL_VAR)GnlNodeSons (Son);  
  
            /* 'Var' is the output of a sequential component */  
            if (GnlVarHook (Var))  
            {  
                SeqCompo = (GNL_SEQUENTIAL_COMPONENT)GnlVarHook (Var);  
                OutputVarBar = GnlSequentialCompoOutputBar (SeqCompo);  
                if (!OutputVarBar)  
                {  
                    sprintf (G_VarBarName, "%s_n", GnlVarName (Var));  
                    if (GnlCreateUniqueVar (Gnl, G_VarBarName,  
                        &OutputVarBar))  
                        return (GNL_MEMORY_FULL);  
                    SetGnlSequentialCompoOutputBar (SeqCompo,  
                        OutputVarBar);  
                    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);  
                    SetGnlVarDir (OutputVarBar, GNL_VAR_LOCAL_WIRING);  
                }  
                if (GnlCreateNodeForVar (Gnl, OutputVarBar, NewNode))  
                    return (GNL_MEMORY_FULL);  
            }  
            return (GNL_OK);  
        }  
    }  
}
```

gnlmapseq.c

```
        }
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        Son = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
        if (GnlCreateOutputVarReplaceInNode (Gnl, Son, &NewSon))
            return (GNL_MEMORY_FULL);
        BListElt (GnlNodeSons (Node), i) = (int) NewSon;
    }

    *NewNode = Node;
    return (GNL_OK);
}

/*-----
/* GnlMapSequentialQBar
/*-----*/
GNL_STATUS GnlMapSequentialQBar (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_COMPONENT ComponentI;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_VAR       OutputVar;
    GNL_FUNCTION  Function;
    GNL_NODE      Node;
    GNL_NODE      NewNode;
    GNL_VAR       VarI;

    GnlResetVarHook (Gnl);

    /* First we scan all the sequential elements with two outputs and */
    /* stores for each output var its corresponding sequential component */
    /* thru the filed 'GnlVarHook'. */
    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
            continue;

        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;

        if (GnlSequentialCompoFormOutput (SeqCompo) != GNL_Q_QBAR)
            continue;

        /* we now that the sequential element we are currently treating */
        /* has two outputs. */
        OutputVar = GnlSequentialCompoOutput (SeqCompo);
        SetGnlVarHook (OutputVar, SeqCompo);
    }

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
```

gnlmapseq.c

```
VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
Function = GnlVarFunction (VarI);
Node = GnlFunctionOnSet (Function);
if (GnlCreateOutputVarReplaceInNode (Gnl, Node, &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlFunctionOnSet (Function, NewNode);
}

return (GNL_OK);
}

/*-----
/* GnlMapSequential                               */
/*-----*/
/* This procedure maps the sequential elements of the current 'Gnl'      */
/* which are Flip-Flops and Latches onto the FF and Latches of the LIBC */
/* library. If the mapping of one cell cannot be done then GNL_MAP_ERROR */
/* is returned, and GNL_OK if everything is ok.                           */
/*-----*/
GNL_STATUS GnlMapSequential (Gnl, GnlLibc)
    GNL          Gnl;
    LIBC_LIB GnlLibc;
{
    BLIST      LibcDffs;
    BLIST      LibcLatches;
    BLIST      GnlDffs;
    BLIST      GnlLatches;
    GNL_STATUS   GnlStatus;
    GNL_LIB     HGnlLib;

    HGnlLib = (GNL_LIB)LibHook (GnlLibc);
    LibcDffs = GnlHLibCellsDffs (HGnlLib);
    LibcLatches = GnlHLibCellsLatches (HGnlLib);

    if (GnlExtractGnlSequential (Gnl, &GnlDffs, &GnlLatches))
        return (GNL_MEMORY_FULL);
    fprintf (stderr, "\n");

/*
    fprintf (stderr, " LIBC: #DFFS = %d\n", BListSize (LibcDffs));
    fprintf (stderr, " LIBC: #LATCHES = %d\n", BListSize (LibcLatches));
    fprintf (stderr, " GNL: #DFFS = %d\n", BListSize (GnlDffs));
    fprintf (stderr, " GNL: #LATCHES = %d\n", BListSize (GnlLatches));
*/
/*
/* We sort flip-flops in the increasing order of their area.           */
qsort (BListAdress (LibcDffs), BListSize (LibcDffs),
       sizeof (LIBC_CELL), CmpLibcCellArea);

/* We sort Latches in the increasing order of their area.                */
qsort (BListAdress (LibcLatches), BListSize (LibcLatches),
       sizeof (LIBC_CELL), CmpLibcCellArea);

/* We map the Gnl flip-flops with the LIBC flip-flops.                  */
*/
```

gnlmapseq.c

```
if ((GnlStatus = GnlMapListDFF (Gnl, GnlDffs, LibcDffs)))
    return (GnlStatus);

/* We map the Gnl latches with the LIBC latches. */
if ((GnlStatus = GnlMapListLATCH (Gnl, GnlLatches, LibcLatches)))
    return (GnlStatus);

/* We eventually map the Q bar pin of the sequential elements */
if ((GnlEnvUseQBar ()) &&
    (GnlStatus = GnlMapSequentialQBar (Gnl)))
    return (GnlStatus);

return (GNL_OK);
}

/*----- EOF -----*/
```

libutil.c

```

/*
 *-----*
 /*      File:          libutil.c           */
 /*      Version:        1.0                */
 /*      Modifications: -                  */
 /*      Documentation: -                  */
 /*                                     */
 /*-----*/
 #include <stdio.h>
#include <malloc.h>
#include "blist.h"
#include "gnl.h"
#include "libc_mem.h"
#include "libc_api.h"

/*-----*
/*----- Global Variable of Timing Analyser -----*/
/* we have difined this global variable because
we consult it always durring the timing analyser */

float G_DeltaProcess;
float G_DeltaTemp;
float G_DeltaVoltage;
LIBC_OPER_COND G_OperCond;
LIBC_WIRE_LOAD G_WireLoad; /* The wire load corresponding a given
                           netlist. This wire_load is choosed in
                           the library taking into account the
                           global area of the netlist.*/ 

/*-----*
/*----- LibGetOperatingConditionsFromName -----*/
/* Doc Procedure LibGetOperatingConditionsFromName :
   - Returns the operating_conditions from name.
   - If the Name is NULL we return the default operating_conditions defined
     in the library.
*/
LIBC_OPER_COND LibGetOperatingConditionsFromName (LIBC_LIB Lib, char *Name)
{
    LIBC_OPER_COND OperCond, OperCondByDefault;

    if (!Name)
        Name = LibDefaultOperatingCond (Lib);

    if (!Name)
        return ( (LIBC_OPER_COND) NULL );

    for (OperCond=LibOperatingCond (Lib); OperCond!=NULL;
                     OperCond=LibOperCondNext(OperCond))
    {
        if (!strcmp (LibOperCondOcName (OperCond), Name))
            return (OperCond);
    }
}

```

libutil.c

```
if (!strcmp (LibOperCondOcName (OperCond), LibDefaultOperatingCond
(Lib)))
    OperCondByDefault = OperCond;
}

/* In the case we do not have any operating_conditions with this Name, we
   return the default operating_conditions defined in the library. */
return (OperCondByDefault);
}

/*-----*
/*----- LibInitializeDeltaOperCondAndNomOperCond -----*/
/* Doc Procedure LibInitializeDeltaOperCondAndNomOperCond :
   - Initialize the globale variables (G_DeltaProcess,
     G_DeltaTemp, G_DeltaVoltage) :
       - G_DeltaProcess = difference of the operating_conditions process
         (choosed by the user or by default) and the nominal process)

       - G_DeltaTemp = difference of the operating_conditions temperature
         (choosed by the user or by default) and the nominal temperature)

       - G_DeltaVoltage = difference of the operating_conditions voltage
         (choosed by the user or by default) and the nominal voltage)
   - Name : name of operating_conditions
*/
void LibInitializeDeltaOperCondAndNomOperCond (LIBC_LIB Lib, char *Name)
{
    LIBC_OPER_COND OperCond, OperCondByDefault;

    G_DeltaProcess = 0.0;
    G_DeltaTemp = 0.0;
    G_DeltaVoltage = 0.0;

    OperCond = LibGetOperatingConditionsFromName (Lib, Name);

    if (!OperCond)
        return;

    G_DeltaProcess = LibOperCondProcess (OperCond) - LibNomProcess (Lib);
    G_DeltaTemp = LibOperCondTemp (OperCond) - LibNomTemp (Lib);
    G_DeltaVoltage = LibOperCondVolt (OperCond) - LibNomVolt (Lib);
    G_OperCond.= OperCond;
}

/*-----*
/*----- LibScalValue -----*/
/* Doc Procedure LibScalValue :
   - Returns the value scaled from a value.
*/
extern float LibScalValue (float          Value,
                           float          KProcess,
                           float          KTemp,
```

```

        float          KVoltage)
{
    float          ValueScaled=0.0;
    float          Process;
    float          Temp;
    float          Voltage;
    float          Delta;

    Process = 1 + G_DeltaProcess * KProcess;
    Temp = 1 + G_DeltaTemp * KTemp;
    Voltage = 1 + G_DeltaVoltage * KVoltage;
    ValueScaled = Value * Process * Temp * Voltage;

    return (ValueScaled);
}
/*-----*
/*----- LibGetPinFromNameAndCell -----*/
/* Doc Procedure LibGetPinFromNameAndCell :
   - Return the Pin (LIBC_PIN) from a Name and a Cell
*/
LIBC_PIN LibGetPinFromNameAndCell (LIBC_CELL Cell, char *PinName)
{
    LIBC_PIN      Pin;

    for (Pin=LibCellPins (Cell); Pin!=NULL; Pin=LibPinNext(Pin))
    {
        if (!GnlSinglePinName (LibPinName (Pin)))
            return ((LIBC_PIN) NULL);

        if (!strcmp (LibPinNameFirst (Pin), PinName))
            return (Pin);
    }

    return ((LIBC_PIN) NULL);
}

/*-----*
/*----- LibGetArcTimingSetupRising -----*/
/* Doc Procedure LibGetArcTimingSetupRising :
   - Return the Timing (LIBC_TIMING) of a given arc (input pin and clock
pin)
   which the type of timing is SetupRising
*/
LIBC_TIMING LibGetArcTimingSetupRising (LIBC_PIN PinIn, LIBC_PIN PinClock)
{
    LIBC_TIMING      Timing;
    LIBC_NAME_LIST  RelatedPin;
}

```

libutil.c

```
for (Timing=LibPinTiming (PinIn); Timing!=NULL;
Timing=LibTimingNext(Timing))
{
    for (RelatedPin=LibTimingRelatedPin (Timing); RelatedPin!=NULL;
         RelatedPin=LibNameListNext(RelatedPin))
    {
        if (!strcmp (LibNameListName (RelatedPin), LibPinNameFirst(PinClock))
&&
            (LibTimingType (Timing) == SETUP_RISING_T) )
            break;
    }

    if (RelatedPin)
        break;
}

return (Timing);
}
/*-----
/*----- LibGetArcTimingSetupFalling -----*/
/* Doc Procedure LibGetArcTimingSetupFalling :
   - Return the Timing (LIBC_TIMING) of a given arc (input pin and clock
pin)
   which the type of timing is SetupFalling
*/
LIBC_TIMING LibGetArcTimingSetupFalling (LIBC_PIN PinIn, LIBC_PIN PinClock)
{
    LIBC_TIMING          Timing;
    LIBC_NAME_LIST    RelatedPin;

    for (Timing=LibPinTiming (PinIn); Timing!=NULL;
Timing=LibTimingNext(Timing))
    {
        for (RelatedPin=LibTimingRelatedPin (Timing); RelatedPin!=NULL;
             RelatedPin=LibNameListNext(RelatedPin))
        {
            if (!strcmp (LibNameListName (RelatedPin), LibPinNameFirst(PinClock))
&&
                (LibTimingType (Timing) == SETUP_FALLING_T) )
                break;
        }

        if (RelatedPin)
            break;
    }

    return (Timing);
}

/*-----
/*----- LibGetArcTiming -----*/
/* Doc Procedure LibGetArcTiming :
   - Return the Timing (LIBC_TIMING) of a given arc (input pin and output
pin)
```

libutil.c

```
/*
LIBC_TIMING LibGetArcTiming (LIBC_PIN PinIn, LIBC_PIN PinOut)
{
    LIBC_TIMING          Timing;
    LIBC_NAME_LIST RelatedPin;

    for (Timing=LibPinTiming (PinOut); Timing!=NULL;
Timing=LibTimingNext(Timing))
    {
        for (RelatedPin=LibTimingRelatedPin (Timing); RelatedPin!=NULL;
              RelatedPin=LibNameListNext(RelatedPin) )
        {
            if (!strcmp (LibNameListName (RelatedPin), LibPinNameFirst(PinIn)))
                break;
        }

        if (RelatedPin)
            break;
    }

    return (Timing);
}

/*-----
/*----- LibInterpolFrom4PointAnd2Values -----*/
/* Doc Procedure LibInterpolFrom4PointAnd2Values :
   - Return the interpolated value from 4 points :
     - (X1, Y1, Z1), (X1, Y2, Z2), (X2, Y1, Z3), (X2, Y2, Z4).
   - Determinate z coordinate ( Z = A + Bx + Cy + Dxy ) and return it
*/
static float LibInterpolFrom4PointAnd2Values (float X1, float X2, float Y1,
                                              float Y2, float Z1, float Z2,
                                              float Z3, float Z4,
                                              float X, float Y)
{
    float A, B, C, D;

    if ( (X1 == X2) && (Y1 == Y2) )
        return (Z1);

    if ( (X1 == X2) && (Y1 != Y2) )
    {
        B = (Z2 - Z1) / (Y2-Y1);
        A = Z1 - B*Y1;
        return ( A + B*Y);
    }

    if ( (X1 != X2) && (Y1 == Y2) )
    {
        B = (Z2 - Z1) / (X2-X1);
        A = Z1 - B*X1;
        return ( A + B*X);
    }
}
```

libutil.c

```
D = (Z4 + Z1 - Z2 - Z3) / ((X2 - X1) * (Y2 - Y1));
C = (Z2 - Z1) / (Y2 - Y1) - D*X1;
B = (Z3 - Z1) / (X2 - X1) - D*Y1;
A = Z1 - B*X1 - C*Y1 - D*X1*Y1;

    return (A + B*X + C*Y + D*X*Y);
}

/*-----*
*----- LibInterpolationFrom2ValuesAndMatrix -----*/
/* Doc Procedure LibInterpolationFrom2ValuesAndMatrix :
   - Return the interpolated value from 2 values and a Matrix of values.
   The dimension of the Matrix is 2.
   For example : - Those 2 values can be capacitance and transtion
                  - The Matrix is cell_rise.
   - Determinate z coordinate ( Z = A + Bx + Cy + Dxy )
*/
float LibInterpolationFrom2ValuesAndMatrix (float X, float Y,
                                             LIBC_TABLE_VAL Matrix)
{
    float_buffer *Index1, *Index2, *Values;
    float X1, X2, Y1, Y2, Z1, Z2, Z3, Z4, Z;
    int I1, I2, J1, J2, Size1, Size2, i;

    Index1 = LibTableValIndex1 (Matrix);
    Index2 = LibTableValIndex2 (Matrix);
    Values = LibTableValValues (Matrix);

    if (!Values)
        return (0.0);
    if ( !Index1 && !Index2)
        return (0.0);

    if (Index1 && !Index2)
    {
        X1 = X2 = 0;
        Size1 = sizeof_float_buffer(Index1);
        if ( Y <= Index1[0] )
        {
            Y1 = Index1[0];
            Y2 = Index1[1];
            J1 = 0;
            J2 = 1;
        }
        else
        if ( Y > Index1[Size1-1] )
        {
            Y1 = Index1[Size1-2];
            Y2 = Index1[Size1-1];
            J1 = Size1-2;
            J2 = Size1-1;
        }
        else
        for (i=0; i < Size1; i++)
        {
```

libutil.c

```
if (Y <= Index1[i])
{
    if (Y < Index1[i])
    {
        Y1 = Index1[i-1];
        Y2 = Index1[i];
        J1 = i-1;
        J2 = i;
    }
    else
    {
        Y1 = Y2 = Y;
        J1 = J2 = i;
    }
    break;
}
Z1 = Values[J1];
Z2 = Values[J2];
Z3 = Z4 = 0.0;
Z = LibInterpolateFrom4PointAnd2Values (X1, X2, Y1, Y2,
                                         Z1, Z2, Z3, Z4, X, Y);
return (Z);
}

Size1 = sizeof_float_buffer(Index1);
Size2 = sizeof_float_buffer(Index2);

if (X <= Index1[0] )
{
    X1 = Index1[0];
    X2 = Index1[1];
    I1 = 0;
    I2 = 1;
}
else
if (X > Index1[Size1-1] )
{
    X1 = Index1[Size1-2];
    X2 = Index1[Size1-1];
    I1 = Size1-2;
    I2 = Size1-1;
}
else
for (i=0; i < Size1; i++)
{
    if (X <= Index1[i])
    {
        if (X < Index1[i])
        {
            X1 = Index1[i-1];
            X2 = Index1[i];
            I1 = i-1;
            I2 = i;
        }
        else
        {
```

libutil.c

```
X1 = X2 = X;
I1 = I2 = i;
}
break;
}

if ( Y <= Index2[0] )
{
    Y1 = Index2[0];
    Y2 = Index2[1];
    J1 = 0;
    J2 = 1;
}
else
if ( Y > Index2[Size2-1] )
{
    Y1 = Index2[Size2-2];
    Y2 = Index2[Size2-1];
    J1 = Size2-2;
    J2 = Size2-1;
}
else
for (i=0; i < Size2; i++)
{
    if (Y <= Index2[i])
    {
        if (Y < Index2[i])
        {
            Y1 = Index2[i-1];
            Y2 = Index2[i];
            J1 = i-1;
            J2 = i;
        }
        else
        {
            Y1 = Y2 = Y;
            J1 = J2 = i;
        }
        break;
    }
}

Z1 = Values[I1*Size2+J1];
Z2 = Values[I1*Size2+J2];
Z3 = Values[I2*Size2+J1];
Z4 = Values[I2*Size2+J2];

Z = LibInterpolateFrom4PointAnd2Values (X1, X2, Y1, Y2, Z1, Z2, Z3, Z4, X, Y);
return (Z);
}

/*-----*/
/*----- LibGetWireLoadSelectFromName -----*/
/* Doc Procedure LibGetWireLoadSelectFromName :
 - Returns the WireLoadSelection from Name and a given Library
```

```

    - if the Name is NULL we return the default_wire_load_selection
*/
LIBC_WIRE_LOAD_SELECT LibGetWireLoadSelectFromName (LIBC_LIB Lib, char *Name)
{
    LIBC_WIRE_LOAD_SELECT          WireLS, WireLSByDefault;

    if (!Name)
        Name = LibDefaultWireLoadSelect (Lib);

    if (!Name)
        return ( (LIBC_WIRE_LOAD_SELECT) NULL);

    for (WireLS=LibWireLoadSelect (Lib); WireLS!=NULL;
                     WireLS=LibWireLoadSelNext(WireLS))
    {
        if (!strcmp (LibWireLoadSelName (WireLS), Name))
            return (WireLS);

        if (!strcmp (LibWireLoadSelName (WireLS), LibDefaultWireLoadSelect
(Lib)))
            WireLSByDefault = WireLS;
    }

    /* In case we do not have any wire_load_selection with this Name */
    return (WireLSByDefault);
}

/*-----*
*----- LibGetWireLoadFromAreaAndWireLS -----*/
/* Doc Procedure LibGetWireLoadFromAreaAndWireLS :
   - Returns the WireLoad from Area and a given wire_load_selection
*/
LIBC_WIRE_LOAD LibGetWireLoadFromAreaAndWireLS (LIBC_WIRE_LOAD_SELECT WireLS,
                                                double Area, LIBC_LIB Lib)
{
    LIBC_WIRE_LOAD_FROM_AREA      WireLFromArea, LastWireLFromArea;
    char                         *Name;
    LIBC_WIRE_LOAD               WireL;

    if (!WireLS)
    {
        Name = LibDefaultWireLoad (Lib);
        if (Name)
        {
            for (WireL=LibWireLoad(Lib); WireL!=NULL;
                  WireL=LibWireLoadNext(WireL))
            {
                if (!strcmp (LibWireLoadName (WireL), Name))
                    return (WireL);
            }
        }
        return (LibWireLoad (Lib));
    }

    for (WireLFromArea=LibWireLoadSelTable (WireLS); WireLFromArea!=NULL;
         WireLFromArea=LibWireLFromAreaNext(WireLFromArea))

```

```

{
    if ( (LibWireLFromAreaMinArea (WireLFromArea) <= Area)
        && (Area <= LibWireLFromAreaMaxArea (WireLFromArea)) )
        return (LibWireLFromAreaLModel (WireLFromArea));
    LastWireLFromArea = WireLFromArea;
}

return (LibWireLFromAreaLModel (LastWireLFromArea));
}

/*-----*
/*----- LibGetWireLengthFromFanout -----*/
/* Doc Procedure LibGetWireLengthFromFanout :
   - Returns the length of the wire_load from fanout
*/
float LibGetWireLengthFromFanout (LIBC_WIRE_LOAD WireL, int Fanout)
{
    LIBC_FAN_LEN      FanLen;
    int              FanoutInf, FanoutSup;
    float             LengthInf, LengthSup, Length;

    if (!WireL)
        return (0.0);

    FanoutInf = FanoutSup = 0;
    LengthInf = LengthSup = 0;

    for (FanLen=LibWireLoadFanLen (WireL); FanLen!=NULL;
                     FanLen=LibFanLenNext (FanLen))
    {
        if (!LibFanLenFanout (FanLen))
            return (0.0);

        if (Fanout == LibFanLenFanout (FanLen))
            return (LibFanLenLength (FanLen));

        if (LibFanLenFanout (FanLen) < Fanout)
        {
            FanoutInf = LibFanLenFanout (FanLen);
            LengthInf = LibFanLenLength (FanLen);
        }
        if (LibFanLenFanout (FanLen) > Fanout && LengthSup == 0)
        {
            FanoutSup = LibFanLenFanout (FanLen);
            LengthSup = LibFanLenLength (FanLen);
            break;
        }
    }

    if (FanoutInf !=0 && LengthSup == 0)
    {
        /* use the slope of wire_load */
        Length = LibWireLoadSlope(WireL) * ((float) (Fanout - FanoutInf));
        return (Length + LengthInf);
    }
    else

```

libutil.c

```
{  
    /* interpolate between FanoutInf and FanoutSup */  
    Length = ((float) (Fanout - FanoutInf)) / ((float) (FanoutSup -  
FanoutInf));  
    Length = Length * (LengthSup - LengthInf);  
    Length = Length + LengthInf;  
    return (Length);  
  
}  
}  
  
/*-----*/  
/*----- LibGetWireLengthFromCapa -----*/  
  
/* Doc Procedure LibGetWireLengthFromCapa :  
   - Returns the length of the wire_load from Capacitance  
*/  
float LibGetWireLengthFromCapa (LIBC_WIRE_LOAD WireL, float Capa)  
{  
    LIBC_FAN_LEN      FanLen;  
    float            CapaInf, CapaSup;  
    float            LengthInf, LengthSup, Length;  
  
    if (!WireL)  
        return (0.0);  
  
    CapaInf = CapaSup = 0.0;  
    LengthInf = LengthSup = 0;  
  
    for (FanLen=LibWireLoadFanLen (WireL); FanLen!=NULL;  
         FanLen=LibFanLenNext (FanLen))  
    {  
        if (!LibFanLenFanout (FanLen))  
            return (0.0);  
  
        if (Capa == LibFanLenCapa (FanLen))  
            return (LibFanLenLength (FanLen));  
  
        if (LibFanLenCapa (FanLen) < Capa)  
        {  
            CapaInf = LibFanLenCapa (FanLen);  
            LengthInf = LibFanLenLength (FanLen);  
        }  
        if (LibFanLenCapa (FanLen) > Capa && LengthSup == 0)  
        {  
            CapaSup = LibFanLenCapa (FanLen);  
            LengthSup = LibFanLenLength (FanLen);  
            break;  
        }  
    }  
  
    if (CapaInf !=0 && LengthSup == 0)  
    {  
        /* use the slope of wire_load */  
        Length = LibWireLoadSlope(WireL) * ((Capa - CapaInf));  
        return (Length + LengthInf);  
    }  
}
```

```

}
else
{
    /* interpolate between CapaInf and CapaSup */
    Length = ((float) (Capa - CapaInf)) / ((float) (CapaSup - CapaInf));
    Length = Length * (LengthSup - LengthInf);
    Length = Length + LengthInf;
    return (Length);
}

/*
*----- LibGetWireScaledCapaFromLength -----
*/
/* Doc Procedure LibGetWireScaledCapaFromLength :
   - Returns the scaled capacitance from the length of a given wire_load
*/
float LibGetWireScaledCapaFromLength (LIBC_WIRE_LOAD WireL, float Length,
                                     LIBC_LIB Lib)
{
    float          Cwire, Capa;
    LIBC_KFATOR    KF;

    if (!WireL)
        return (0.0);

    Capa = LibWireLoadCapa(WireL);
    if (Capa == 0.0)
        Capa = DefaultWireLoadCapa(Lib);

    Cwire = Capa * Length;
    KF = LibKFactor (Lib);

    if (KF)
        Cwire = LibScalValue (Cwire, LibKFactorWireCap(KF) [0],
                             LibKFactorWireCap(KF) [1],
                             LibKFactorWireCap(KF) [2]);

    return (Cwire);
}

/*
*----- LibGetWireScaledResiFromLength -----
*/
/* Doc Procedure LibGetWireScaledResiFromLength :
   - Returns the scaled resistance from the length of a given wire_load
*/
float LibGetWireScaledResiFromLength (LIBC_WIRE_LOAD WireL, float Length,
                                      LIBC_LIB Lib)
{
    float          ResiWire, Resi;
    LIBC_KFATOR    KF;

    if (!WireL)

```

libutil.c

```
return (0.0);

Resi = LibWireLoadResistance(WireL);
if (Resi == 0)
    Resi = LibDefaultWireLoadResistance(Lib);

ResiWire = Resi * Length;
KF = LibKFactor (Lib);
if (KF)
    ResiWire = LibScalValue (ResiWire, LibKFactorWireRes(KF) [0],
                           LibKFactorWireRes(KF) [1],
                           LibKFactorWireRes(KF) [2]);

return (ResiWire);
}

/*-----
/*----- LibGetWireAreaFromLength -----*/
/* Doc Procedure LibGetWireAreaFromLength :
   - Returns the area from the length of a given wire_load
*/
float LibGetWireAreaFromLength (LIBC_WIRE_LOAD WireL, float Length,
                               LIBC_LIB Lib)
{
    float          Area;

    if (!WireL)
        return (0.0);

    Area = LibWireLoadArea(WireL);
    if (Area == 0.0)
        Area = LibDefaultWireLoadArea(Lib);

    return (Area * Length);
}

/*-----
/*----- */
/*----- */
/*----- */

/*----- LibGetScaledValFromTransCapaAndMatrix -----*/
/* Doc Procedure LibGetScaledValFromTransCapaAndMatrix :
   - Return the value corresponding to a transition "Trans" and capacitance
     "Capa" in a given matrix (cell_rise, cell_fall, rise_propagation,
     fall_propagation, rise_transition, fall_transition, rise_constraint,
     fall_constraint). This matrix is from the cell "Cell".
   - This Value is scaled taking into account the scaling_factors in the
cell
     "Cell"
*/
float LibGetScaledValFromTransCapaAndMatrix (float Trans, float Capa,
                                             LIBC_TABLE_VAL Matrix,
```

```

LIBC_CELL Cell,
LIB_MATRIX_TYPE Type)
{
float Val;
float Process, Temp, Volt;

if (!Matrix)
    return (0.0);

Val = LibInterpolationFrom2ValuesAndMatrix (Trans, Capa, Matrix);

if (!LibCellScalingFactors (Cell))
    return (Val);

switch (Type)
{
case MATRIX_CELL_RISE:
    Process = LibKFactorCellRise (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorCellRise (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorCellRise (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_CELL_FALL:
    Process = LibKFactorCellFall (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorCellFall (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorCellFall (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_RISE_PROPAGATION:
    Process = LibKFactorRisePropagation (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorRisePropagation (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorRisePropagation (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_FALL_PROPAGATION:
    Process = LibKFactorFallPropagation (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorFallPropagation (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorFallPropagation (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_RISE_TRANSITION:
    Process = LibKFactorRiseTransition (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorRiseTransition (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorRiseTransition (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_FALL_TRANSITION:
    Process = LibKFactorFallTransition (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorFallTransition (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorFallTransition (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_SETUP_RISE:
    Process = LibKFactorSetupRise (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorSetupRise (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorSetupRise (LibCellScalingFactors (Cell))[2];
    break;
case MATRIX_SETUP_FALL:
    Process = LibKFactorSetupFall (LibCellScalingFactors (Cell))[0];
    Temp = LibKFactorSetupFall (LibCellScalingFactors (Cell))[1];
    Volt = LibKFactorSetupFall (LibCellScalingFactors (Cell))[2];
    break;
}

Val = LibScalValue (Val, Process, Temp, Volt);

```

libutil.c

```
    return (Val);
}

/*-----*
/*----- LibDelayArcCellRise -----*/
/* Doc Procedure LibDelayArcCellRise :
 - Compute the delay arc rising "DelayR" and the transition arc rising
   "OutTransR".
   The arce is (PinIn, PinOut) from the cell "Cell"
   OutCapa : the globale capacitance (the total of capacitace of all pins
   connected to the output pin "PinOutName" + the capa of the wire
connected
   to the output); it is scaled.
 - The values (inTransR, InTransF, OutCapa) are scaled.
 - This wire delay is not included.
*/
void LibDelayArcCellRise (float InTransR, float InTransF, float OutCapa,
                         LIBC_CELL Cell, LIBC_PIN PinIn, LIBC_PIN PinOut,
                         float *DelayR, float *OutTransR)
{
    LIBC_TIMING          Timing;
    LIB_MATRIX_TYPE      Type;
    LIBC_TABLE_VAL       Matrix;
    float                Val1, Val2;

    Timing = LibGetArcTiming (PinIn, PinOut);

    if (!Timing)
    {
        *DelayR = *OutTransR = 0.0;
        return;
    }

    Matrix = LibTimingRisePropagation (Timing);
    Type = MATRIX_CELL_RISE;
    if (Matrix)
        Type = MATRIX_RISE_PROPAGATION;

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *OutTransR = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                               LibTimingRiseTransition (Timing),
                                                               Cell, MATRIX_RISE_TRANSITION);
            if (Type == MATRIX_RISE_PROPAGATION)
            {
                Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                               LibTimingRisePropagation (Timing),
                                                               Cell, MATRIX_RISE_PROPAGATION);
                *DelayR = Val1 + *OutTransR;
            }
            else
                *DelayR = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                               LibTimingCellRise (Timing),
                                                               Cell, MATRIX_CELL_RISE);
    }
}
```

```

        break;
    case NEGATIVE_UNATE_E:
        *OutTransR = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
            LibTimingRiseTransition (Timing),
            Cell, MATRIX_RISE_TRANSITION);
        if (Type == MATRIX_RISE_PROPAGATION)
        {
            Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                LibTimingRisePropagation (Timing),
                Cell, MATRIX_RISE_PROPAGATION);
            *DelayR = Val1 + *OutTransR;
        }
        else
            *DelayR = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                LibTimingCellRise (Timing),
                Cell, MATRIX_CELL_RISE);
        break;

    case NON_UNATE_E:
        Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
            LibTimingRiseTransition (Timing),
            Cell, MATRIX_RISE_TRANSITION);
        Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
            LibTimingRiseTransition (Timing),
            Cell, MATRIX_RISE_TRANSITION);
        *OutTransR = Val1;
        if (*OutTransR < Val2)
            *OutTransR = Val2;

        if (Type == MATRIX_RISE_PROPAGATION)
        {
            Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                LibTimingRisePropagation (Timing),
                Cell, MATRIX_RISE_PROPAGATION);
            Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                LibTimingRisePropagation (Timing),
                Cell, MATRIX_RISE_PROPAGATION);
            *DelayR = Val1 + *OutTransR;
            if (Val1 < Val2)
                *DelayR = Val2 + *OutTransR;
        }
        else
        {
            Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                LibTimingCellRise (Timing),
                Cell, MATRIX_CELL_RISE);
            Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                LibTimingCellRise (Timing),
                Cell, MATRIX_CELL_RISE);
            *DelayR = Val1;
            if (Val1 < Val2)
                *DelayR = Val2;
        }
        break;
    }
}

```

```

/*
 *-----*
 *----- LibDelayArcCellRiseSetup -----*
 */

/* Doc Procedure LibDelayArcCellRiseSetup :
 - Compute the delay arc rising "DelayR" .
 The arc is (PinIn, PinOut) from the sequentiel cell "Cell"
 OutCapa : the globale capacitance (the total of capacitace of all pins
 connected to the the pin "PinInName" + the capa of the wire connected
 to the pin); it is scaled.
 - The values (inTransR, InTransF, OutCapa) are scaled.
 - This wire delay is not included.
 */
void LibDelayArcCellRiseSetup (float InTransR, float InTransF, float OutCapa,
                               LIBC_CELL Cell, LIBC_PIN PinIn, LIBC_PIN PinClock,
                               float *DelayR)
{
    LIBC_TIMING          Timing;
    float                Val1, Val2;

    Timing = LibGetArcTimingSetupRising (PinIn, PinClock);

    if (!Timing)
    {
        Timing = LibGetArcTimingSetupFalling (PinIn, PinClock);
        if (!Timing)
        {
            *DelayR = 0.0;
            return;
        }
    }

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *DelayR = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                               LibTimingRiseConstraint (Timing),
                                                               Cell, MATRIX_SETUP_RISE);
            break;
        case NEGATIVE_UNATE_E:
            *DelayR = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                               LibTimingRiseConstraint (Timing),
                                                               Cell, MATRIX_SETUP_RISE);
            break;
        case NON_UNATE_E:
            Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                          LibTimingRiseConstraint (Timing),
                                                          Cell, MATRIX_SETUP_RISE);
            Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                          LibTimingRiseConstraint (Timing),
                                                          Cell, MATRIX_SETUP_RISE);
            *DelayR = Val1;
            if (Val1 < Val2)
                *DelayR = Val2;
            break;
    }
}

```

```

}

}

/*-----
----- LibDelayArcCellFallSetup -----*/
/* Doc Procedure LibDelayArcCellFallSetup :
 - Compute the delay arc falling "DelayF" .
 The arc is (PinIn, PinOut) from the sequentiel cell "Cell"
 OutCapa : the globale capacitance (the total of capacitace of all pins
 connected to the the pin "PinInName" + the capa of the wire connected
 to the pin); it is scaled.
 - The values (inTransR, InTransF, OutCapa) are scaled.
 - This wire delay is not included.
*/
void LibDelayArcCellFallSetup (float InTransR, float InTransF, float OutCapa,
                             LIBC_CELL Cell, LIBC_PIN PinIn, LIBC_PIN PinClock,
                             float *DelayF)
{
    LIBC_TIMING          Timing;
    float                 Val1, Val2;

    Timing = LibGetArcTimingSetupFalling (PinIn, PinClock);

    if (!Timing)
    {
        Timing = LibGetArcTimingSetupRising (PinIn, PinClock);
        if (!Timing)
        {
            *DelayF = 0.0;
            return;
        }
    }

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *DelayF = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                               LibTimingFallConstraint (Timing),
                                                               Cell, MATRIX_SETUP_FALL);
            break;
        case NEGATIVE_UNATE_E:
            *DelayF = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                               LibTimingFallConstraint (Timing),
                                                               Cell, MATRIX_SETUP_FALL);
            break;

        case NON_UNATE_E:
            Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                          LibTimingFallConstraint (Timing),
                                                          Cell, MATRIX_SETUP_FALL);
            Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                          LibTimingFallConstraint (Timing),
                                                          Cell, MATRIX_SETUP_FALL);
            *DelayF = Val1;
            if (Val1 < Val2)

```

```

        *DelayF = Val2;

        break;
    }

}

/*-----*
/*----- LibDelayArcCellFall -----*/
/* Doc Procedure LibDelayArcCellFall :
   - Compute the delay arc rising "DelayF" and the transition arc rising
     "OutTransF".
   The arce is (PinInName, PinOutName) from the cell "Cell"
   OutCapa : the globale capacitance (the total of capacitace of all pins
   connected to the output pin "PinOutName" + the capa of the wire
connected
   to the output); it is scaled.
   - The values (inTransR, InTransF, OutCapa) are scaled.
   - The wire delay is not included.
*/
void LibDelayArcCellFall (float InTransR, float InTransF, float OutCapa,
                         LIBC_CELL Cell, LIBC_PIN PinIn, LIBC_PIN PinOut,
                         float *DelayF, float *OutTransF)
{
    LIBC_TIMING          Timing;
    LIB_MATRIX_TYPE      Type;
    LIBC_TABLE_VAL       Matrix;
    float                Vall, Val2;

    Timing = LibGetArcTiming (PinIn, PinOut);

    if (!Timing)
    {
        *DelayF = *OutTransF = 0.0;
        return;
    }

    Matrix = LibTimingFallPropagation (Timing);
    Type = MATRIX_CELL_FALL;
    if (Matrix)
        Type = MATRIX_FALL_PROPAGATION;

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *OutTransF = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                               LibTimingFallTransition (Timing),
                                                               Cell, MATRIX_FALL_TRANSITION);
            if (Type == MATRIX_FALL_PROPAGATION)
            {
                Vall = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                               LibTimingFallPropagation (Timing),
                                                               Cell, MATRIX_FALL_PROPAGATION);
                *DelayF = Vall + *OutTransF;
            }
        else

```

```

*DelayF = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                LibTimingCellFall (Timing),
                                                Cell, MATRIX_CELL_FALL);
break;
case NEGATIVE_UNATE_E:
*OutTransF = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                LibTimingFallTransition (Timing),
                                                Cell, MATRIX_FALL_TRANSITION);
if (Type == MATRIX_FALL_PROPAGATION)
{
    Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                LibTimingFallPropagation (Timing),
                                                Cell, MATRIX_FALL_PROPAGATION);
    *DelayF = Val1 + *OutTransF;
}
else
    *DelayF = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                LibTimingCellFall (Timing),
                                                Cell, MATRIX_CELL_FALL);
break;

case NON_UNATE_E:
Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                              LibTimingFallTransition (Timing),
                                              Cell, MATRIX_FALL_TRANSITION);
Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                              LibTimingFallTransition (Timing),
                                              Cell, MATRIX_FALL_TRANSITION);
*OutTransF = Val1;
if (*OutTransF < Val2)
    *OutTransF = Val2;

if (Type == MATRIX_FALL_PROPAGATION)
{
    Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                LibTimingFallPropagation (Timing),
                                                Cell, MATRIX_FALL_PROPAGATION);
    Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                LibTimingFallPropagation (Timing),
                                                Cell, MATRIX_FALL_PROPAGATION);
    *DelayF = Val1 + *OutTransF;
    if (Val1 < Val2)
        *DelayF = Val2 + *OutTransF;
}
else
{
    Val1 = LibGetScaledValFromTransCapaAndMatrix (InTransR, OutCapa,
                                                LibTimingCellFall (Timing),
                                                Cell, MATRIX_CELL_FALL);
    Val2 = LibGetScaledValFromTransCapaAndMatrix (InTransF, OutCapa,
                                                LibTimingCellFall (Timing),
                                                Cell, MATRIX_CELL_FALL);
    *DelayF = Val1;
    if (Val1 < Val2)
        *DelayF = Val2;
}
break;

```

```

}

}

/*-----*/
/*----- LibDelayWire -----*/
/* Doc Procedure LibDelayWire :
 - Compute the delay of a wire with capacitance "OutCapa" and resistance
 "Resistance"
 OutCapa : the globale capacitance (the total of capacitace of all pins
 connected to the output pin "PinOutName" + the capa of the wire;
 it is scaled.
*/
void LibDelayWire (float OutCapa, float Resistance, float *WireDelay,
                   int Fanout)
{
    enum tree_type_E      OperCondTreeType;

    OperCondTreeType = LibOperCondTreeType (G_OperCond);
    switch (OperCondTreeType)
    {
        case BEST_CASE_TREE :
            *WireDelay = 0.0;
            break;
        case BALANCED_TREE :
            *WireDelay = Resistance/Fanout * (OutCapa/Fanout);
            break;
        case WORST_CASE_TREE :
            *WireDelay = Resistance * OutCapa;
            break;
    }
}
/*----- LibDelayArcCell -----*/
/* Doc Procedure LibDelayArcCell :
 - Compute :
     . DelayR : the delay arc rising
     . DelayF : the delay arc falling
     . OutTransR : the transition arc rising
     . OutTransF : the transition arc rising
 The arce is (PinInName, PinOutName) from the cell "Cell"
 OutCapa : the globale capacitance (the total of capacitace of all pins
 connected to the output pin "PinOutName" + the capa of the wire
 connected
 to the output); it is scaled.
 - The values (inTransR, InTransF, OutCapa) are scaled.
 - The wire delay is included.
*/
void LibDelayArcCell (float InTransR, float InTransF, float OutCapa,
                      float WireResistance, int Fanout,
                      LIBC_CELL Cell, LIBC_PIN PinIn, LIBC_PIN PinOut,
                      float *DelayR, float *OutTransR,
                      float *DelayF, float *OutTransF)
{
    float      WireDelay;

```

libutil.c

```
LibDelayArcCellRise (InTransR, InTransF, OutCapa, Cell, PinIn, PinOut,
                      DelayR, OutTransR);
LibDelayArcCellFall (InTransR, InTransF, OutCapa, Cell, PinIn, PinOut,
                      DelayF, OutTransF);
WireDelay = 0.0;
if ((*DelayR) || (*DelayF))
    LibDelayWire (OutCapa, WireResistance, &WireDelay, Fanout);

*OutTransR = *OutTransR + WireDelay;
*OutTransF = *OutTransF + WireDelay;
*DelayR = *DelayR + WireDelay;
*DelayF = *DelayF + WireDelay;

}

extern void GnlGetOutputPinFromPins ();

/*-----EOF-----*/
```

libutil.e

```
/*-----*/
/*                                         */
/*      File:          libutil.e           */
/*      Version:        1.0                */
/*      Modifications: -                  */
/*      Documentation: -                  */
/*                                         */
/*-----*/
/*-----*/
extern LIBC_OPER_COND LibGetOperatingConditionsFromName (LIBC_LIB Lib, char *Name);

extern void LibInitializeDeltaOperCondAndNomOperCond (LIBC_LIB Lib, char *Name);

extern float LibScalValue (float      Value,
                           float      KProcess,
                           float      KTemp,
                           float      KVoltage);

extern LIBC_PIN LibGetPinFromNameAndCell (LIBC_CELL Cell, char *PinName);

extern LIBC_TIMING LibGetArcTiming (LIBC_PIN PinIn, LIBC_PIN PinOut);

extern float LibGetScaledValFromTransCapaAndMatrix (float Trans, float Capa,
                                                    LIBC_TABLE_VAL Matrix,
                                                    LIBC_CELL Cell,
                                                    LIB_MATRIX_TYPE Type);

extern LIBC_WIRE_LOAD_SELECT LibGetWireLoadSelectFromName (LIBC_LIB Lib, char *Name);

extern LIBC_WIRE_LOAD LibGetWireLoadFromAreaAndWireLS (LIBC_WIRE_LOAD_SELECT WireLS, double Area, LIBC_LIB Lib);

extern float LibGetWireLengthFromFanout (LIBC_WIRE_LOAD WireL, int Fanout);

extern float LibGetWireLengthFromCapa (LIBC_WIRE_LOAD WireL, float Capa);

extern float LibGetWireScaledCapaFromLength (LIBC_WIRE_LOAD WireL,
                                              float Length, LIBC_LIB Lib);

extern float LibGetWireScaledResiFromLength (LIBC_WIRE_LOAD WireL,
                                              float Length, LIBC_LIB Lib);

extern float LibGetWireAreaFromLength (LIBC_WIRE_LOAD WireL, float Length,
                                       LIBC_LIB Lib);

extern void LibDelayArcCellRise (float InTransR, float InTransF, float OutCapa,
                                 LIBC_CELL Cell, char *PinInName, char *PinOutName,
                                 float *DelayR, float *OutTransR);

extern void LibDelayArcCellFall (float InTransR, float InTransF, float OutCapa,
                                 LIBC_CELL Cell, char *PinInName, char *PinOutName,
                                 float *DelayF, float *OutTransF);
```

libutil.e

```
extern void LibDelayArcCell (float InTransR, float InTransF, float OutCapa,
                            float WireResistance, int Fanout,
                            LIBC_CELL Cell, LIBC_PIN PinInName, LIBC_PIN PinOutName,
                            float *DelayR, float *OutTransR,
                            float *DelayF, float *OutTransF);

extern void LibDelayArcCellRiseSetup (float InTransR, float InTransF,
                                      float OutCapa, LIBC_CELL Cell, LIBC_PIN PinIn,
                                      LIBC_PIN PinClock, float *DelayR);

extern void LibDelayArcCellFallSetup (float InTransR, float InTransF,
                                      float OutCapa, LIBC_CELL Cell, LIBC_PIN PinIn,
                                      LIBC_PIN PinClock, float *DelayF);

extern GNL_STATUS LibGetListOfBufferInLib (LIBC_LIB Lib, BLIST *ListBuffers);

/*-----EOF-----*/
```

list.txt

Volume in drive D is UNICHATSRC
Volume Serial Number is 051E-4A11

Directory of d:\sourceuni

11/28/99	03:33p	2,950	CHA.BMP
11/28/99	03:33p	129,446	Man-test.bmp
11/28/99	03:33p	96,450	MAN00.BMP
11/28/99	03:33p	308,278	sprite000.bmp
11/28/99	03:33p	19,314	Test-uni.BMP
11/28/99	03:33p	308,278	TEST0.BMP
11/28/99	03:33p	308,278	TESTMAP.BMP

7 File(s) 1,172,994 bytes

Directory of d:\sourceuni\MonTool

11/28/99	03:32p	306	AWAY.BMP
11/28/99	03:32p	290	CHAT.BMP
11/28/99	03:32p	302	CHATPRIV.BMP
11/28/99	03:32p	254	FOLDER.BMP
11/28/99	03:32p	334	HOST.BMP
11/28/99	03:32p	126	MEMBER.BMP
11/28/99	03:32p	5,804	MSN.BMP
11/28/99	03:32p	290	PART.BMP
11/28/99	03:32p	286	SENDBAR.BMP
11/28/99	03:32p	292	SPEC.BMP
11/28/99	03:32p	1,116	TOOLBAR.BMP
11/28/99	03:32p	312	WORLD.BMP

12 File(s) 9,712 bytes

Directory of d:\sourceuni\UDSGen

11/28/99	03:32p	12,672	MBtnLan.bmp
11/28/99	03:32p	14,328	MBtnMdm.bmp
11/28/99	03:32p	8,952	MBtnNo.bmp
11/28/99	03:32p	8,820	MBtnOk.bmp
11/28/99	03:32p	34,944	MLgnAni.bmp
11/28/99	03:32p	133,380	MLoginBk.bmp

6 File(s) 213,096 bytes

Total Files Listed:

25 File(s) 1,395,802 bytes
0 bytes free



APPENDIX C

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

gnlarea.c

```
/*
 *-----*/
/*      File:          gnlarea.c
 *      Version:        1.1
 *      Modifications: -
 *      Documentation: -
 */
/*      Description:          */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnlestim.h"
#include "time.h"
#include "bbdd.h"
#include "gnllibc.h"
#include "gnlmap.h"

#include "blist.e"
#include "libutil.e"
#include "timeutil.e"
#include "timecomp.e"

/*
 *-----*/
/* GnlGetAreaWithoutNetFromGnl          */
/*-----*/
GNL_STATUS GnlGetAreaWithoutNetFromGnl (GNL      Gnl,
                                         double   *CombArea, double *NonCombArea,
                                         int     *CombCellNumber,
                                         int     *NonCombCellNumber,
                                         int     *BlackBoxNumber)
{
    int                  i;
    GNL_COMPONENT        Component;
    GNL                  GnlCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_BUF_COMPONENT    BufCompo;
    GNL_USER_COMPONENT   UserCompo;
    GNL_STATUS           GnlStatus;
    LIBC_CELL             Cell;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        Component = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        switch (GnlComponentType (Component)) {
```

```

case GNL_SEQUENTIAL_COMPO:
    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) Component;
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    *NonCombArea += LibCellArea (Cell);
    (*NonCombCellNumber)++;
    break;

case GNL_USER_COMPO:
    UserCompo = (GNL_USER_COMPONENT) Component;
    GnlCompo = GnlUserComponentGnlDef (UserCompo);
    if (GnlCompo)
    {
        if (GnlStatus = GnlGetAreaWithoutNetFromGnl (GnlCompo,
CombArea,
                           NonCombArea, CombCellNumber,
                           NonCombCellNumber, BlackBoxNumber))
            return (GnlStatus);
        continue;
    }
    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
    if (Cell)
    {
        (*CombCellNumber)++;
        *CombArea += LibCellArea (Cell);
    }
    else
        (*BlackBoxNumber)++;
    break;

case GNL_TRISTATE_COMPO:
    TriStateCompo = (GNL_TRISTATE_COMPONENT) Component;
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);
    *NonCombArea += LibCellArea (Cell);
    (*NonCombCellNumber)++;
    break;

case GNL_BUF_COMPO:
    BufCompo = (GNL_BUF_COMPONENT) Component;
    Cell = GnlBufCompoInfoCell (BufCompo);
    *NonCombArea += LibCellArea (Cell);
    (*NonCombCellNumber)++;
    break;

case GNL_MACRO_COMPO:
    break;

default:
    GnlError (12 /* Unknown component */);
    break;
}

return (GNL_OK);
}

/*-----
/* GnlGetNetInterconnetAreaFromGnlComp
 */

```

gnlarea.c

```
/*-----*/
GNL_STATUS GnlGetNetInterconnetAreaFromGnlComp (GNL_COMPONENT Compo,
                                                LIBC_LIB Lib,
                                                LIBC_WIRE_LOAD WireLoad,
                                                double      *NetArea)
{
    LIBC_CELL          Cell;
    int               i, Fanout, Rank;
    LIBC_PIN          PinOut;
    GNL_VAR           VarI;
    GNL_ASSOC         AssocI;
    double            Length, SingleArea;
    BLIST             Interface;
    char              *Formal;
    GNL_STATUS        GnlStatus;
    GNL_TIMING_INFO  TimingI;
    unsigned int      Key;

    GnlGetCellFromGnlCompo (Compo, &Cell);
    GnlGetInterfaceFromGnlCompo (Compo, &Interface);

    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if (!AssocI)
            continue;
        VarI = GnlAssocActualPort (AssocI);
        if (!VarI)
            continue;
        if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
            continue;
        Formal = (char *) GnlAssocFormalPort (AssocI);
        if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
            continue;
        if (LibPinDirection(PinOut) == INPUT_E)
            continue;

        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                                    &TimingI, &Key, &Rank)))
            return (GnlStatus);
        Fanout = GnlTimingInfoFanout (TimingI);
        Length = LibGetWireLengthFromFanout (WireLoad, Fanout);
        SingleArea = LibGetWireAreaFromLength (WireLoad, Length, Lib);
        (*NetArea) += SingleArea;
    }
    return (GNL_OK);
}

/*-----*/
/* GnlGetNetInterconnetAreaFromGnl                               */
/*-----*/
GNL_STATUS GnlGetNetInterconnetAreaFromGnl (Gnl, Lib, WireLoad, NetArea,
                                             NetNumber)
GNL           Gnl;
LIBC_LIB      Lib;
LIBC_WIRE_LOAD WireLoad;
```

gnlarea.c

```
double          *NetArea;
int             *NetNumber;
{
    int             i;
    GNL_COMPONENT   Component;
    GNL            GnlCompo;
    GNL_USER_COMPONENT UserCompo;
    GNL_STATUS     GnlStatus;
    LIBC_CELL       Cell;

*NetNumber += BListSize (GnlInputs (Gnl));
*NetNumber += BListSize (GnlLocals (Gnl));
*NetNumber += BListSize (GnlOutputs (Gnl));

for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
{
    Component = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
    switch (GnlComponentType (Component)) {
        case GNL_SEQUENTIAL_COMPO:
        case GNL_TRISTATE_COMPO:
        case GNL_BUF_COMPO:
            if (GnlStatus = GnlGetNetInterconnetAreaFromGnlComp (Component,
                                                       Lib, WireLoad, NetArea))
                return (GnlStatus);
            break;

        case GNL_USER_COMPO:
            UserCompo = (GNL_USER_COMPONENT) Component;
            GnlCompo = GnlUserComponentGnlDef (UserCompo);
            if (GnlCompo)
            {
                if (BListAddElt (G_PileOfComponent, UserCompo))
                    return (GNL_MEMORY_FULL);
                if (GnlStatus = GnlGetNetInterconnetAreaFromGnl (GnlCompo,
Lib,
                                                       WireLoad, NetArea, NetNumber))
                    return (GnlStatus);
                BListDelShift (G_PileOfComponent, BListSize
(G_PileOfComponent));
                continue;
            }
            Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
            if (Cell)
                if (GnlStatus = GnlGetNetInterconnetAreaFromGnlComp
(Component,
                                                       Lib, WireLoad, NetArea))
                    return (GnlStatus);
            break;

        case GNL_MACRO_COMPO:
            break;

        default:
            GnlError (12 /* Unknown component */);
            break;
    }
}
```

gnlarea.c

```
}

SetGnlNetArea (Gnl, *NetArea);

return (GNL_OK);
}

/*-----
/* GnlPrintHeadOfAreaReport                               */
/*-----*/
/* Doc procedure GnlPrintHeadOfAreaReport :             */
/* - Printing global parameters used for timing analysis */
*/
/*-----*/
GNL_STATUS GnlPrintHeadOfAreaReport (GNL Gnl,
                                      LIBC_LIB     Lib,
                                      double        CombArea,
                                      double        NonCombArea,
                                      double        NetArea,
                                      int          CellNumber,
                                      int          NetNumber)
{

fprintf (stderr, " ----- \n");
fprintf (stderr, " Report\t:\tarea\n");
fprintf (stderr, " Design\t:\t%s\n", GnlName (Gnl));
fprintf (stderr, " Version\t:\t1999.01\n");
fprintf (stderr, " Date\t:\t\t");
GnlPrintDate (stderr);
fprintf (stderr, "\n");

fprintf (stderr, " Library Used\t:\t%s\n\n", LibName (Lib));

fprintf (stderr, " Number Of Ports\t:\t%d\n", GnlNbIn (Gnl) + GnlNbOut (Gnl));
fprintf (stderr, " Number Of Cells\t:\t%d\n", CellNumber);

fprintf (stderr, " Combinational Area\t:\t%.2f\n", CombArea);
fprintf (stderr, " Non Combinational Area:\t%.2f\n", NonCombArea);
fprintf (stderr, " Net Interconnect Area\t:\t%.2f\n", NetArea);

fprintf (stderr, " Total Area\t:\t%.2f\n", CombArea+NonCombArea+NetArea);
fprintf (stderr, " ----- \n");

return (GNL_OK);
}

/*-----
/* GnlGetAreaFromNetwork                                */
/*-----*/
/* This procedure extracts the global area (area of generic cells */
/* (Combinational & nonCombinational) + area of Net Interconnect). */
*/
/*-----*/
GNL_STATUS GnlGetAreaFromNetwork (GNL_NETWORK Nw,
```

```

LIBC_LIB Lib,
        int PrintData,
double *AreaWithoutNet,
double *AreaOfNets,
int *CellNumber)

{
    GNL TopGnl;
    GNL_STATUS GnlStatus;
    int i;
    int CombCellNumber;
    int NonCombCellNumber;
    int BlackBoxNumber;
    int NetNumber;
    double CombArea, NonCombArea;
    double AreaNetwork;
    double NetArea;
    LIBC_WIRE_LOAD_SELECT WireLoadSelect;

TopGnl = GnlNetworkTopGnl (Nw);

if (BListCreate (&G_PileOfComponent))
    return (GNL_MEMORY_FULL);

CombArea = NonCombArea = 0.0;
CombCellNumber = NonCombCellNumber = BlackBoxNumber = 0;
if ((GnlStatus = GnlGetAreaWithoutNetFromGnl (TopGnl, &CombArea,
                                              &NonCombArea, &CombCellNumber,
                                              &NonCombCellNumber, &BlackBoxNumber)))
    return (GnlStatus);

AreaNetwork = CombArea + NonCombArea;
*AreaWithoutNet = AreaNetwork;
WireLoadSelect = LibGetWireLoadSelectFromName (Lib, (char *)NULL);
G_WireLoad = LibGetWireLoadFromAreaAndWireLS (WireLoadSelect, AreaNetwork,
                                              Lib);

NetArea = 0.0;
NetNumber = 0;
if ((GnlStatus = GnlGetNetInterconnetAreaFromGnl (TopGnl, Lib,
                                                 G_WireLoad, &NetArea, &NetNumber)))
    return (GnlStatus);

*AreaOfNets = NetArea;
*CellNumber = NonCombCellNumber+CombCellNumber;

if (PrintData)
{
    if (GnlStatus = GnlPrintHeadOfAreaReport (TopGnl, Lib, CombArea,
                                              NonCombArea, NetArea,
                                              NonCombCellNumber+CombCellNumber,
                                              NetNumber))
        return (GnlStatus);
}

BListQuickDelete (&G_PileOfComponent);
return (GNL_OK);
}

```

gnlarea.c

```
}
```

```
/*-----*/
```

```
/* ----- EOF ----- */
```

gnlarea.e

```
/*-----*/
/*
 *      File:          gnlarea.e
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description:          */
/*
/*-----*/
extern GNL_STATUS GnlGetAreaFromNetwork (GNL_NETWORK  Nw,
                                         LIBC_LIB     Lib,
                                         int          PrintData,
                                         double       *AreaWithoutNet,
                                         double       *AreaOfNets);

/*
----- EOF -----*/
/*-----*/
```

File: gnlarea.e

gnlbench.c

```
/*-----*/  
/*-----*/  
/* File: gnlbench.c */  
/* Version: 1.1 */  
/* Modifications: - */  
/* Documentation: - */  
/*-----*/  
/* Description: this file defines functions in order to generate a */  
/* a verilog test bench corresponding to a verilog */  
/* netlist. First the netlist is analyzed and then */  
/* specific signals like Clocks and Asynchronous Set */  
/* and reset are extracted and instantiate in an */  
/* intelligent way (e.g clocks are instantiated like */  
/* classical clocks and set and reset are activated */  
/* firts at their activated level in order to put the */  
/* state machine in a Set/Reset mode. This is to start */  
/* Hdl simulation in a realistic way. */  
/* Enhancements can be performed when instantiating */  
/* stimuli for primary inputs. For now there are gene- */  
/* rated in an exhaustive manner in the function */  
/* 'GnlComputeNextValue'. To improve the simulation */  
/* cover one can re-write this function in order to */  
/* find better stimuli. */  
/*-----*/  
/*-----*/
```

```
#include <stdio.h>  
  
#ifdef MEMORY /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
  
#include "blist.e"  
  
/*-----*/  
/* DEFINE */  
/*-----*/  
#define GnlVarValue(v) ((int)((v)->Hook))  
#define SetGnlVarValue(v,s) ((((v)->Hook = (int*)s))  
  
/*-----*/  
/* GnlExtractClocks */  
/*-----*/  
/* Scan all the sequential elements and adds all the clock signals */  
/* in the returned list 'ListClocks'. A clock signal must be a simple */  
/* signal originally defined by the user (e.g not expanded). */  
/*-----*/  
GNL_STATUS GnlExtractClocks (Gnl, ListClocks)  
{  
    GNL Gnl;  
    BLIST *ListClocks;  
    Components;  
    int i;
```

gnlbench.c

```
GNL_SEQUENTIAL_COMPONENT SequentialCompo;
GNL_COMPONENT ComponentI;
GNL_VAR ClockVar;

if (BListCreate (&ListClocks))
    return (GNL_MEMORY_FULL);

Components = GnlComponents (Gnl);

if (!Components)
    return (GNL_OK);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT) BListElt (Components, i);
    if (GnlComponentType (ComponentI) == GNL_SEQUENTIAL_COMPO)
    {
        SequentialCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
        if ((GnlSequentialCompoOp (SequentialCompo) == GNL_DFF) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFFX) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFF0) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFF1))
        {
            ClockVar = GnlSequentialCompoClock (SequentialCompo);
            if (GnlVarRangeSize (ClockVar) != 1)
            {
                GnlError (11 /* Clock signal must be a simple signal */);
                return (GNL_BAD_CLOCK_DEFINITION);
            }
            if (GnlVarType (ClockVar) == GNL_VAR_EXPANSED)
                continue;
            if (!BListMemberOfList ((*&ListClocks), ClockVar,
                                   IntIdentical))
                if (BListAddElt ((*&ListClocks), ClockVar))
                    return (GNL_MEMORY_FULL);
        }
    }
}

return (GNL_OK);
}

/*
 * GnlExtractAsynchSignals
 */
/*
 * Analyze DFF components and extracts list of asynchronous signals
 * (Set/reset) which are active at Low/High level. A signal cannot be
 * part of ListAsynchHigh and ListAsynchLow at the same time.
 */
GNL_STATUS GnlExtractAsynchSignals (Gnl, ListAsynchHigh, ListAsynchLow)
    GNL          Gnl;
    BLIST        *ListAsynchHigh;
    BLIST        *ListAsynchLow;
{
    BLIST        Components;
```

```

int i;
GNL_SEQUENTIAL_COMPONENT SequentialCompo;
GNL_COMPONENT ComponentI;

if (BListCreate (ListAsynchHigh))
    return (GNL_MEMORY_FULL);

if (BListCreate (ListAsynchLow))
    return (GNL_MEMORY_FULL);

Components = GnlComponents (Gnl);

if (!Components)
    return (GNL_OK);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) == GNL_SEQUENTIAL_COMPO)
    {
        SequentialCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
        if ((GnlSequentialCompoOp (SequentialCompo) == GNL_DFF) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFFX) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFF0) ||
            (GnlSequentialCompoOp (SequentialCompo) == GNL_DFF1))
        {
            if (GnlVarIsSignal (
                GnlSequentialCompoReset (SequentialCompo)) &&
                !BListMemberOfList ((*ListAsynchHigh),
                    GnlSequentialCompoReset (SequentialCompo),
                    IntIdentical) &&
                !BListMemberOfList ((*ListAsynchLow),
                    GnlSequentialCompoReset (SequentialCompo),
                    IntIdentical))
            )
                if (BListAddElt ((*ListAsynchHigh),
                    GnlSequentialCompoReset (SequentialCompo)))
                    return (GNL_MEMORY_FULL);
        }
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlExtractInputs */
/*-----*/
/* Extracts the list of primary inputs and scan the internal Hash Table */
/* names to do so. Primary inputs must be of type GNL_VAR_ORIGINAL e.g. */
/* Variables defined by the user and not internally expanded. */
/*-----*/
GNL_STATUS GnlExtractInputs (Gnl, ListClocks, ListAsynchLow,
                            ListAsynchHigh, ListInputs)
GNL Gnl;
BLIST ListClocks;

```

gnlbench.c

```
BLIST      ListAsynchLow;
BLIST      ListAsynchHigh;
BLIST      *ListInputs;
{
    int          i;
    int          j;
    BLIST      ListI;
    GNL_VAR     VarJ;
    BLIST      HashTableNames;

    if (BListCreate (ListInputs))
        return (GNL_MEMORY_FULL);

    /* We scan the HashTableNames in order to pick up primary input and */
    /* inout signals. We take only GNL_VAR_ORIGINAL signals.           */
    HashTableNames = GnlHashNames (Gnl);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        ListI = (BLIST)BListElt (HashTableNames, i);
        if (ListI)
        {
            for (j=0; j<BListSize (ListI); j++)
            {
                VarJ = (GNL_VAR)BListElt (ListI, j);
                if (((GnlVarDir (VarJ) == GNL_VAR_INPUT) ||
                     (GnlVarDir (VarJ) == GNL_VAR_INOUT)) &&
                    (GnlVarType (VarJ) == GNL_VAR_ORIGINAL) &&
                    !BListMemberOfList (ListAsynchHigh, VarJ, IntIdentical) &&
                    !BListMemberOfList (ListAsynchLow, VarJ, IntIdentical) &&
                    !BListMemberOfList (ListClocks, VarJ, IntIdentical))
                    if (BListAddElt ((*ListInputs), (int)VarJ))
                        return (GNL_MEMORY_FULL);
            }
        }
    }

    return (GNL_OK);
}

/*
 * GnlExtractOutputs
 */
/*
 * Extracts the list of primary outputs and scan the internal Hash Table*/
/* names to do so. Primary outputs must be of type GNL_VAR_ORIGINAL e.g.*/
/* Variables defined by the user and not internally expanded.           */
/*
 *-----*/
GNL_STATUS GnlExtractOutputs (Gnl, ListClocks, ListAsynchLow,
                               ListAsynchHigh, ListOutputs)
{
    GNL          Gnl;
    BLIST        ListClocks;
    BLIST        ListAsynchLow;
    BLIST        ListAsynchHigh;
    BLIST        *ListOutputs;
{
    int          i;
    int          j;
```

gnlbench.c

```
BLIST      ListI;
GNL_VAR    VarJ;
BLIST      HashTableNames;

if (BListCreate (ListOutputs))
    return (GNL_MEMORY_FULL);

/* We scan the HashTableNames in order to pick up primary input and */
/* inout signals. We take only GNL_VAR_ORIGINAL signals.           */
HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    ListI = (BLIST)BListElt (HashTableNames, i);
    if (ListI)
    {
        for (j=0; j<BListSize (ListI); j++)
        {
            VarJ = (GNL_VAR)BListElt (ListI, j);
            if (((GnlVarDir (VarJ) == GNL_VAR_OUTPUT) ||
                  (GnlVarDir (VarJ) == GNL_VAR_INOUT)) &&
                (GnlVarType (VarJ) == GNL_VAR_ORIGINAL) &&
                !BListMemberOfList (ListAsynchHigh, VarJ, IntIdentical) &&
                !BListMemberOfList (ListAsynchLow, VarJ, IntIdentical) &&
                !BListMemberOfList (ListClocks, VarJ, IntIdentical))
                if (BListAddElt ((*ListOutputs), (int)VarJ))
                    return (GNL_MEMORY_FULL);
        }
    }
}

return (GNL_OK);
}

/*
 * GnlPrintHeaderbench
 */
/*
 * Prints official header in the verilog test bench.
 */
void GnlPrintHeaderbench (benchFile)
FILE          *benchFile;
{
    fprintf (benchFile,
             "/*-----\n");
    fprintf (benchFile,
             "/* This file has been automatically generated by GNL\n");
    fprintf (benchFile,
             "/*-----\n");
    fprintf (benchFile,
             "/* Copyright (c) 1998 by interHDL, Inc.\n");
    fprintf (benchFile,
             "/*-----\n");
}

/*
 */

/*-----*/
```

gnlbench.c

```
/* GnlPrintModuleAndInterface */  
/*-----*/  
GNL_STATUS GnlPrintModuleAndInterface (benchFile, GnlName, ListClocks,  
                                         ListInputs, ListOutputs,  
                                         ListAsyncLow, ListAsyncHigh)  
FILE          *benchFile;  
char           *GnlName;  
BLIST          ListClocks;  
BLIST          ListInputs;  
BLIST          ListOutputs;  
BLIST          ListAsyncLow;  
BLIST          ListAsyncHigh;  
{  
    int           i;  
    GNL_VAR      ClockI;  
    GNL_VAR      InputI;  
    GNL_VAR      OutputI;  
    char          *RangeStr;  
  
    fprintf (benchFile, "module %s_top () ;\n", GnlName);  
  
    if (BListSize (ListClocks))  
        fprintf (benchFile, "// Clock signals\n");  
    for (i=0; i<BListSize (ListClocks); i++)  
    {  
        ClockI = (GNL_VAR)BListElt (ListClocks, i);  
        fprintf (benchFile, "wire %s;\n", GnlVarName (ClockI));  
        fprintf (benchFile, "reg %s_sh;\n", GnlVarName (ClockI));  
    }  
    fprintf (benchFile, "\n");  
  
    if (BListSize (ListInputs))  
        fprintf (benchFile, "// Input signals\n");  
    for (i=0; i<BListSize (ListInputs); i++)  
    {  
        InputI = (GNL_VAR)BListElt (ListInputs, i);  
        if (GnlVarGetStrFromRange (InputI, &RangeStr))  
            return (GNL_MEMORY_FULL);  
  
        if (RangeStr)  
            fprintf (benchFile, "reg %s %s;\n", RangeStr,  
                    GnlVarName (InputI));  
        else  
            fprintf (benchFile, "reg %s;\n", GnlVarName (InputI));  
    }  
    fprintf (benchFile, "\n");  
  
    if (BListSize (ListOutputs))  
        fprintf (benchFile, "// Output signals\n");  
    for (i=0; i<BListSize (ListOutputs); i++)  
    {  
        OutputI = (GNL_VAR)BListElt (ListOutputs, i);  
        if (GnlVarGetStrFromRange (OutputI, &RangeStr))  
            return (GNL_MEMORY_FULL);  
  
        if (RangeStr)
```

gnlbench.c

```
        fprintf (benchFile, "wire %s %s;\n", RangeStr,
                  GnlVarName (OutputI));
    else
        fprintf (benchFile, "wire %s;\n", GnlVarName (OutputI));
    }
fprintf (benchFile, "\n");

if (BListSize (ListAsyncLow))
    fprintf (benchFile, "// Asynchronous signals (Active Low Level)\n");
for (i=0; i<BListSize (ListAsyncLow); i++)
{
    OutputI = (GNL_VAR)BListElt (ListAsyncLow, i);

    if (GnlVarType (OutputI) == GNL_VAR_ORIGINAL)
    {
        if (GnlVarGetStrFromRange (OutputI, &RangeStr))
            return (GNL_MEMORY_FULL);

        if (RangeStr)
            fprintf (benchFile, "reg %s %s;\n", RangeStr,
                      GnlVarName (OutputI));
        else
            fprintf (benchFile, "reg %s;\n", GnlVarName (OutputI));
    }
}
fprintf (benchFile, "\n");

if (BListSize (ListAsyncHigh))
    fprintf (benchFile, "// Asynchronous signals (Active High Level)\n");
for (i=0; i<BListSize (ListAsyncHigh); i++)
{
    OutputI = (GNL_VAR)BListElt (ListAsyncHigh, i);

    if (GnlVarType (OutputI) == GNL_VAR_ORIGINAL)
    {
        if (GnlVarGetStrFromRange (OutputI, &RangeStr))
            return (GNL_MEMORY_FULL);

        if (RangeStr)
            fprintf (benchFile, "reg %s %s;\n", RangeStr,
                      GnlVarName (OutputI));
        else
            fprintf (benchFile, "reg %s;\n", GnlVarName (OutputI));
    }
}
fprintf (benchFile, "\n");

fprintf (benchFile, "\n");

return (GNL_OK);
}

/*-----
/* GnlPrintInitial
/*-----*/
```

gnlbench.c

```
void GnlPrintInitial (benchFile, ListClocks, TimeLimit)
FILE *benchFile;
BLIST ListClocks;
int TimeLimit;
{
    int i;
    GNL_VAR ClockI;

    fprintf (benchFile, "initial\n");
    fprintf (benchFile, "begin\n");
    fprintf (benchFile, "$dumpvars;\n");

    for (i=0; i<BListSize (ListClocks); i++)
    {
        ClockI = (GNL_VAR)BListElt (ListClocks, i);
        fprintf (benchFile, "%s_sh = 0;\n", GnlVarName (ClockI));
    }
    fprintf (benchFile, "#%d\n", TimeLimit);
    fprintf (benchFile, "$finish;\n");
    fprintf (benchFile, "end\n\n");
}

/*-----
/* GnlPrintAlwaysClocks
*/
void GnlPrintAlwaysClocks (benchFile, ListClocks, Period)
FILE *benchFile;
BLIST ListClocks;
int Period;
{
    int i;
    GNL_VAR ClockI;

    for (i=0; i<BListSize (ListClocks); i++)
    {
        ClockI = (GNL_VAR)BListElt (ListClocks, i);
        fprintf (benchFile, "assign %s = %s_sh;\n", GnlVarName (ClockI),
                 GnlVarName (ClockI));
    }
    fprintf (benchFile, "\n");

    for (i=0; i<BListSize (ListClocks); i++)
    {
        ClockI = (GNL_VAR)BListElt (ListClocks, i);
        fprintf (benchFile, "always\n");
        fprintf (benchFile, "begin\n");
        fprintf (benchFile, "#%d %s_sh = ~%s_sh;\n", Period,
                 GnlVarName (ClockI), GnlVarName (ClockI));
        fprintf (benchFile, "end\n\n");
    }
}

/*-----
/* GnlModifyWithAsyncSignals
*/
-----*/
```

gnlbench.c

```
GNL_STATUS GnlModifyWithAsyncSignals (Var, Range, StrBin, SetResetMode,
                                      ListAsyncLow, ListAsyncHigh)
{
    GNL_VAR Var;
    int          Range;
    char         *StrBin;
    int          SetResetMode;
    BLIST       ListAsyncLow;
    BLIST       ListAsyncHigh;
{
    int          i;
    GNL_VAR AsyncI;
    char         *VarName;
    int          Index;

    for (i=0; i<BListSize (ListAsyncLow); i++)
    {
        AsyncI = (GNL_VAR)BListElt (ListAsyncLow, i);
        if (GnlVarStrNameStrIndex (GnlVarName (AsyncI), &VarName, &Index))
            return (GNL_MEMORY_FULL);

        if (Index == -1)           /* simple signal */           */
            Index = 0;

        if (!strcmp (GnlVarName (Var), VarName))
        {
            if (SetResetMode)
                StrBin[Range-Index-1] = '0';      /* active at low level */
            else
                StrBin[Range-Index-1] = '1';      /* SetResetMode disable */
        }

        free (VarName);
    }

    for (i=0; i<BListSize (ListAsyncHigh); i++)
    {
        AsyncI = (GNL_VAR)BListElt (ListAsyncHigh, i);
        if (GnlVarStrNameStrIndex (GnlVarName (AsyncI), &VarName, &Index))
            return (GNL_MEMORY_FULL);

        if (Index == -1)           /* simple signal */           */
            Index = 0;

        if (!strcmp (GnlVarName (Var), VarName))
        {
            if (SetResetMode)
                StrBin[Range-Index-1] = '1';      /* active at high level */
            else
                StrBin[Range-Index-1] = '0';      /* SetResetMode disable */
        }

        free (VarName);
    }
    return (GNL_OK);
}
```

gnlbench.c

```
/*-----*/
/* GnlComputeNextValue                               */
/*-----*/
/* ENHANCEMENT: one can rewrite this part in order to assign more */
/* 'intelligent' patterns allowing a best cover of the netlist. One      */
/* will need 'Gnl' in order to compute those stimuli.                      */
/*-----*/
GNL_STATUS GnlComputeNextValue (Gnl, Var, Range)
    GNL          Gnl;
    GNL_VAR     Var;
    int         Range;
{
    /* Computation of next value is trivial: simply increment it.  */
    SetGnlVarValue (Var, ((int)GnlVarValue (Var)+(int)1));

    return (GNL_OK);
}

/*-----*/
/* GnlPrintAlwaysInputs                            */
/*-----*/
GNL_STATUS GnlPrintAlwaysInputs (benchFile, Gnl, ListInputs, ListAsyncLow,
                                ListAsyncHigh, TimeLimit, Step)
    FILE        *benchFile;
    GNL          Gnl;
    BLIST       ListInputs;
    BLIST       ListAsyncLow;
    BLIST       ListAsyncHigh;
    int         TimeLimit;
    int         Step;
{
    int         Time = 0;
    int         i;
    GNL_VAR    InputI;
    int         RangeSize;
    char        *StrBin;
    int         SetResetMode;

    for (i=0; i<BListSize (ListInputs); i++)
    {
        InputI = (GNL_VAR)BListElt (ListInputs, i);
        SetGnlVarValue (InputI, 0);
    }
    for (i=0; i<BListSize (ListAsyncLow); i++)
    {
        InputI = (GNL_VAR)BListElt (ListAsyncLow, i);
        SetGnlVarValue (InputI, 0);
    }
    for (i=0; i<BListSize (ListAsyncHigh); i++)
    {
        InputI = (GNL_VAR)BListElt (ListAsyncHigh, i);
        SetGnlVarValue (InputI, 0);
    }

    SetResetMode = 1;      /* We start in Set/Reset mode
                           */
}
```

```

fprintf (benchFile, "always\n");
fprintf (benchFile, "    begin\n");
while (Time <= TimeLimit)
{
    if (Time != 0)
    {
        fprintf (benchFile, "#%d\n", Step);

        /* We are not in set/reset mode any longer */ 
        SetResetMode = 0;
    }

    /* At time 'Time', we assign patterns to primary inputs */
    for (i=0; i<BListSize (ListInputs); i++)
    {
        InputI = (GNL_VAR)BListElt (ListInputs, i);
        RangeSize = GnlVarRangeSize (InputI);

        /* Compute the next Value of InputI and modifies field */
        /* GnlVarValue (InputI). */
        if (GnlComputeNextValue (Gnl, InputI, RangeSize))
            return (GNL_MEMORY_FULL);

        /* Translate the GnlVarValue (InputI) into string of length*/
        /* RangeSize. */
        if (GnlDecimalToStrBin (GnlVarValue (InputI), RangeSize,
                               &StrBin))
            return (GNL_MEMORY_FULL);

        /* We need to modify the previous pattern in order to take */
        /* into account asynchronous signals. */
        if (GnlModifyWithAsyncSignals (InputI, RangeSize,
                                      StrBin, SetResetMode,
                                      ListAsyncLow, ListAsyncHigh))
            return (GNL_MEMORY_FULL);

        fprintf (benchFile, "%s = %d'b%s;\n",
                GnlVarName (InputI), RangeSize, StrBin);

        free (StrBin);
    }

    /* At time 'Time', we assign patterns to primary inputs */
    for (i=0; i<BListSize (ListAsyncLow); i++)
    {
        InputI = (GNL_VAR)BListElt (ListAsyncLow, i);
        if (GnlVarType (InputI) == GNL_VAR_ORIGINAL)
        {
            RangeSize = GnlVarRangeSize (InputI);

            /* Compute the next Value of InputI and modifies field*/
            /* GnlVarValue (InputI). */
            if (GnlComputeNextValue (Gnl, InputI, RangeSize))
                return (GNL_MEMORY_FULL);

            /* Translate the GnlVarValue (InputI) into string of */

```

```

/* length RangeSize. */  

if (GnlDecimalToStrBin (GnlVarValue (InputI), RangeSize,  

                        &StrBin))  

    return (GNL_MEMORY_FULL);  
  

/* We need to modify the previous pattern in order to */  

/* take into account asynchronous signals. */  

if (GnlModifyWithAsyncSignals (InputI, RangeSize,  

                            StrBin, SetResetMode,  

                            ListAsyncLow, ListAsyncHigh))  

    return (GNL_MEMORY_FULL);  
  

fprintf (benchFile, "      %s = %d\b%s;\n",
        GnlVarName (InputI), RangeSize, StrBin);  
  

free (StrBin);  
  

}  

}  
  

/* At time 'Time', we assign patterns to primary inputs */  

for (i=0; i<BListSize (ListAsyncHigh); i++)  

{
    InputI = (GNL_VAR)BListElt (ListAsyncHigh, i);
    if (GnlVarType (InputI) == GNL_VAR_ORIGINAL)
    {
        RangeSize = GnlVarRangeSize (InputI);  
  

        /* Compute the next Value of InputI and modifies field*/  

        /* GnlVarValue (InputI). */  

        if (GnlComputeNextValue (Gnl, InputI, RangeSize))
            return (GNL_MEMORY_FULL);  
  

        /* Translate the GnlVarValue (InputI) into string of */  

        /* length RangeSize. */  

        if (GnlDecimalToStrBin (GnlVarValue (InputI), RangeSize,
                               &StrBin))
            return (GNL_MEMORY_FULL);  
  

        /* We need to modify the previous pattern in order to */  

        /* take into account asynchronous signals. */  

        if (GnlModifyWithAsyncSignals (InputI, RangeSize,
                                    StrBin, SetResetMode,
                                    ListAsyncLow, ListAsyncHigh))
            return (GNL_MEMORY_FULL);  
  

        fprintf (benchFile, "      %s = %d\b%s;\n",
                GnlVarName (InputI), RangeSize, StrBin);  
  

        free (StrBin);  

    }
}
}

```

gnlbench.c

```
        Time += Step;
    }

    fprintf (benchFile, "    end\n\n");
    return (GNL_OK);
}

/*-----*/
/* GnlPrintAlwaysTrace */
/*-----*/
GNL_STATUS GnlPrintAlwaysTrace (benchFile, ListClocks, ListOutPuts,
ListInputs,
                                ListAsyncLow, ListAsyncHigh)
FILE          *benchFile;
BLIST          ListClocks;
BLIST          ListOutPuts;
BLIST          ListInputs;
BLIST          ListAsyncLow;
BLIST          ListAsyncHigh;
{
    int           i;
    GNL_VAR      ClockI;
    GNL_VAR      OutputI;
    GNL_VAR      InputI;
    GNL_VAR      VarI;
    BLIST         ListAsync;
    int           OnePrint = 0;

    /* Extracting the original async signals of the netlist */
    if (BListCreate (&ListAsync))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListAsyncLow); i++)
    {
        VarI = (GNL_VAR)BListElt (ListAsyncLow, i);
        if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
            if (BListAddElt (ListAsync, (int*)VarI))
                return (GNL_MEMORY_FULL);
    }

    for (i=0; i<BListSize (ListAsyncHigh); i++)
    {
        VarI = (GNL_VAR)BListElt (ListAsyncHigh, i);
        if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
            if (BListAddElt (ListAsync, (int*)VarI))
                return (GNL_MEMORY_FULL);
    }

    fprintf (benchFile, "always @( ");

    for (i=0; i<BListSize (ListClocks)-1; i++)
    {
        ClockI = (GNL_VAR)BListElt (ListClocks, i);
        fprintf (benchFile, "%s_sh or ", GnlVarName (ClockI));
        OnePrint++;
    }
```

gnlbench.c

```
if (BListSize (ListClocks))
{
    ClockI = (GNL_VAR)BListElt (ListClocks, i);
    fprintf (benchFile, "%s_sh ", GnlVarName (ClockI));
    OnePrint++;
}

if (BListSize (ListInputs) && OnePrint)
    fprintf (benchFile, "or ");

for (i=0; i<BListSize (ListOutPuts)-1; i++)
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, "%s or ", GnlVarName (OutputI));
    OnePrint++;
}
if (BListSize (ListOutPuts))
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, "%s ", GnlVarName (OutputI));
    OnePrint++;
}

if (BListSize (ListInputs) && OnePrint)
    fprintf (benchFile, "or ");

for (i=0; i<BListSize (ListInputs)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, "%s or ", GnlVarName (InputI));
    OnePrint++;
}
if (BListSize (ListInputs))
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, "%s ", GnlVarName (InputI));
    OnePrint++;
}

if (BListSize (ListAsync) && OnePrint)
    fprintf (benchFile, "or ");

for (i=0; i<BListSize (ListAsync)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, "%s or ", GnlVarName (InputI));
    OnePrint++;
}
if (BListSize (ListAsync))
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, "%s ", GnlVarName (InputI));
    OnePrint++;
}

fprintf (benchFile, ")\n");
```

gnlbench.c

```
fprintf (benchFile, "    begin\n");
fprintf (benchFile, "        $display(\"time = %cd, \", '%');

OnePrint = 0;

for (i=0; i<BListSize (ListClocks)-1; i++)
{
    ClockI = (GNL_VAR)BListElt (ListClocks, i);
    fprintf (benchFile, "%s_sh = %cd,", GnlVarName (ClockI), '%');
    OnePrint++;
}
if (BListSize (ListClocks))
{
    ClockI = (GNL_VAR)BListElt (ListClocks, i);
    fprintf (benchFile, "%s_sh = %cd", GnlVarName (ClockI), '%');
    OnePrint++;
}

if (BListSize (ListOutPuts) && OnePrint)
    fprintf (benchFile, ", ");

for (i=0; i<BListSize (ListOutPuts)-1; i++)
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, "%s = %cd,", GnlVarName (OutputI), '%');
    OnePrint++;
}
if (BListSize (ListOutPuts))
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, "%s = %cd", GnlVarName (OutputI), '%');
    OnePrint++;
}

if (BListSize (ListInputs) && OnePrint)
    fprintf (benchFile, ", ");

for (i=0; i<BListSize (ListInputs)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, "%s = %cd,", GnlVarName (InputI), '%');
    OnePrint++;
}
if (BListSize (ListInputs))
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, "%s = %cd", GnlVarName (InputI), '%');
    OnePrint++;
}

if (BListSize (ListAsync) && OnePrint)
    fprintf (benchFile, ", ");

for (i=0; i<BListSize (ListAsync)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, "%s = %cd,", GnlVarName (InputI), '%');
```

gnlbench.c

```
    OnePrint++;
}
if (BListSize (ListAsync))
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, "%s = %cd", GnlVarName (InputI), '%');
    OnePrint++;
}

OnePrint = 0;
fprintf (benchFile, "\\\n\\", $time,");

for (i=0; i<BListSize (ListClocks)-1; i++)
{
    ClockI = (GNL_VAR)BListElt (ListClocks, i);
    fprintf (benchFile, " %s_sh,", GnlVarName (ClockI));
    OnePrint++;
}
if (BListSize (ListClocks))
{
    ClockI = (GNL_VAR)BListElt (ListClocks, i);
    fprintf (benchFile, " %s_sh", GnlVarName (ClockI));
    OnePrint++;
}

if (BListSize (ListOutPuts) && OnePrint)
    fprintf (benchFile, ",");

for (i=0; i<BListSize (ListOutPuts)-1; i++)
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, " %s,", GnlVarName (OutputI));
    OnePrint++;
}
if (BListSize (ListOutPuts))
{
    OutputI = (GNL_VAR)BListElt (ListOutPuts, i);
    fprintf (benchFile, " %s", GnlVarName (OutputI));
    OnePrint++;
}

if (BListSize (ListInputs) && OnePrint)
    fprintf (benchFile, ",");

for (i=0; i<BListSize (ListInputs)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, " %s,", GnlVarName (InputI));
    OnePrint++;
}
if (BListSize (ListInputs))
{
    InputI = (GNL_VAR)BListElt (ListInputs, i);
    fprintf (benchFile, " %s", GnlVarName (InputI));
    OnePrint++;
}
```

gnlbench.c

```
if (BListSize (ListAsync) && OnePrint)
    fprintf (benchFile, ",");

for (i=0; i<BListSize (ListAsync)-1; i++)
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, " %s,", GnlVarName (InputI));
    OnePrint++;
}
if (BListSize (ListAsync))
{
    InputI = (GNL_VAR)BListElt (ListAsync, i);
    fprintf (benchFile, " %s", GnlVarName (InputI));
    OnePrint++;
}

BListQuickDelete (&ListAsync);

fprintf (benchFile, ";\n");
fprintf (benchFile, "    end\n\n");

return (GNL_OK);
}

/*-----
/* GnlPrintInstanceAndEndModule
*/
-----*/
GNL_STATUS GnlPrintInstanceAndEndModule (benchFile, GnlName, ListInputs,
                                         ListOutputs, ListClocks,
                                         ListAsyncLow, ListAsyncHigh)
FILE          *benchFile;
char          *GnlName;
BLIST         ListInputs;
BLIST         ListOutputs;
BLIST         ListClocks;
BLIST         ListAsyncLow;
BLIST         ListAsyncHigh;
{
    int          MaxId = 0;
    int          i;
    int          j;
    GNL_VAR     VarI;
    BLIST        ListAsync;

    /* Extracting the original async signals of the netlist */
    if (BListCreate (&ListAsync))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListAsyncLow); i++)
    {
        VarI = (GNL_VAR)BListElt (ListAsyncLow, i);
        if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
            if (BListAddElt (ListAsync, (int*)VarI))
                return (GNL_MEMORY_FULL);
    }
}
```

gnlbench.c

```
for (i=0; i<BListSize (ListAsyncHigh); i++)
{
    VarI = (GNL_VAR)BListElt (ListAsyncHigh, i);
    if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
        if (BListAddElt (ListAsync, (int*)VarI))
            return (GNL_MEMORY_FULL);
}

fprintf (benchFile, "%s U1 (", GnlName);

for (i=0; i<BListSize (ListInputs); i++)
{
    VarI = (GNL_VAR)BListElt (ListInputs, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListOutputs); i++)
{
    VarI = (GNL_VAR)BListElt (ListOutputs, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListClocks); i++)
{
    VarI = (GNL_VAR)BListElt (ListClocks, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListAsync); i++)
{
    VarI = (GNL_VAR)BListElt (ListAsync, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}

for (j=0; j< MaxId; j++)
{
    for (i=0; i<BListSize (ListInputs); i++)
    {
        VarI = (GNL_VAR)BListElt (ListInputs, i);
        if (GnlVarId (VarI) == j)
            fprintf (benchFile, "%s, ", GnlVarName (VarI));
    }
    for (i=0; i<BListSize (ListOutputs); i++)
    {
        VarI = (GNL_VAR)BListElt (ListOutputs, i);
        if (GnlVarId (VarI) == j)
            fprintf (benchFile, "%s, ", GnlVarName (VarI));
    }
    for (i=0; i<BListSize (ListClocks); i++)
    {
        VarI = (GNL_VAR)BListElt (ListClocks, i);
        if (GnlVarId (VarI) == j)
            fprintf (benchFile, "%s, ", GnlVarName (VarI));
    }
    for (i=0; i<BListSize (ListAsync); i++)
```

gnlbench.c

```
{  
    VarI = (GNL_VAR)BListElt (ListAsync, i);  
    if (GnlVarId (VarI) == j)  
        fprintf (benchFile, "%s, ", GnlVarName (VarI));  
}  
}  
  
for (i=0; i<BListSize (ListInputs); i++)  
{  
    VarI = (GNL_VAR)BListElt (ListInputs, i);  
    if (GnlVarId (VarI) == j)  
        fprintf (benchFile, "%s", GnlVarName (VarI));  
}  
for (i=0; i<BListSize (ListOutputs); i++)  
{  
    VarI = (GNL_VAR)BListElt (ListOutputs, i);  
    if (GnlVarId (VarI) == j)  
        fprintf (benchFile, "%s", GnlVarName (VarI));  
}  
for (i=0; i<BListSize (ListClocks); i++)  
{  
    VarI = (GNL_VAR)BListElt (ListClocks, i);  
    if (GnlVarId (VarI) == j)  
        fprintf (benchFile, "%s", GnlVarName (VarI));  
}  
for (i=0; i<BListSize (ListAsync); i++)  
{  
    VarI = (GNL_VAR)BListElt (ListAsync, i);  
    if (GnlVarId (VarI) == j)  
        fprintf (benchFile, "%s", GnlVarName (VarI));  
}  
  
fprintf (benchFile, ");\n\n");  
fprintf (benchFile, "endmodule\n");  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGenerateBenchFile */  
/*-----*/  
GNL_STATUS GnlGenerateBenchFile (Gnl, ListClocks, ListInputs,  
                                ListOutputs, ListAsyncLow, ListAsyncHigh,  
                                TimeLimit, Period, Step)  
{  
    GNL          Gnl;  
    BLIST        ListClocks;  
    BLIST        ListInputs;  
    BLIST        ListOutputs;  
    BLIST        ListAsyncLow;  
    BLIST        ListAsyncHigh;  
    int          TimeLimit;  
    int          Period;  
    int          Step;  
    FILE         *benchFile;  
    char         *NewName;
```

gnlbench.c

```
char *GnlName;

GnlName = GnlName (Gnl);

if (GnlStrAppendStrCopy (GnlName, "_bench.v", &NewName))
    return (GNL_MEMORY_FULL);

if ((benchFile = fopen (NewName, "w")) == NULL)
    return (GNL_CANNOT_OPEN_OUTFILE);

fprintf (stderr, "# Generating test bench file : [%s]\n", NewName);

GnlPrintHeaderbench (benchFile);

if (GnlPrintModuleAndInterface (benchFile, GnlName, ListClocks,
                               ListInputs, ListOutputs,
                               ListAsyncLow, ListAsyncHigh))
    return (GNL_MEMORY_FULL);

GnlPrintInitial (benchFile, ListClocks, TimeLimit);

GnlPrintAlwaysClocks (benchFile, ListClocks, Period);

if (GnlPrintAlwaysInputs (benchFile, Gnl, ListInputs, ListAsyncLow,
                         ListAsyncHigh, TimeLimit, Step))
    return (GNL_MEMORY_FULL);

if (GnlPrintAlwaysTrace (benchFile, ListClocks, ListOutputs, ListInputs,
                        ListAsyncLow, ListAsyncHigh))
    return (GNL_MEMORY_FULL);

if (GnlPrintInstanceAndEndModule (benchFile, GnlName, ListInputs,
                                 ListOutputs, ListClocks,
                                 ListAsyncLow, ListAsyncHigh))
    return (GNL_MEMORY_FULL);

fclose (benchFile);

return (GNL_OK);
}

/*-----*/
/* GnlGenerateBench */
/*-----*/
GNL_STATUS GnlGenerateBench (Gnl, TimeLimit, Period, Step)
    GNL      Gnl;
    int      TimeLimit;
    int      Period;
    int      Step;
{
    GNL_STATUS      GnlStatus;
    BLIST      ListClocks;
    BLIST      ListInputs;
    BLIST      ListOutputs;
```

gnlbench.c

```
BLIST      ListAsynchHigh;
BLIST      ListAsynchLow;

if ((GnlStatus = GnlExtractClocks (Gnl, &ListClocks)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractAsynchSignals (Gnl,
                                         &ListAsynchHigh,
                                         &ListAsynchLow)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractInputs (Gnl, ListClocks, ListAsynchLow,
                                   ListAsynchHigh, &ListInputs)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractOutputs (Gnl, ListClocks, ListAsynchLow,
                                   ListAsynchHigh, &ListOutputs)))
    return (GnlStatus);

GnlGenerateBenchFile (Gnl, ListClocks, ListInputs,
                      ListOutputs, ListAsynchLow, ListAsynchHigh,
                      TimeLimit, Period, Step);

    return (GNL_OK);
}

/*-----*/
```

gnlcheck.c

```
/*-----*/  
/*-----*/  
/* File: gnlcheck.c */  
/* Version: 1.1 */  
/* Modifications: - */  
/* Documentation: - */  
/*-----*/  
/* Description: */  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
  
#include "blist.e"  
  
#define MAX_DEPTH_CYCLE 20000  
  
/*-----*/  
/* Global Variables */  
/*-----*/  
static BLIST G_NodeStack;  
static BLIST G_NodesOnCycle;  
  
/*-----*/  
/* GnlMaybeCycle */  
/*-----*/  
/* This procedures return the following values for 'Res': */  
/* 0 --> Leaves from node 'Node' have a depth less than */  
/* MAX_DEPTH_CYCLE so there is no potential cycle */  
/* 1 --> A leaf with depth = MAX_DEPTH_CYCLE has been */  
/* reached and the list 'ListCycle' is currently build */  
/* corresponding to the recursive calls of 'GnlMaybeCycle' */  
/* on each 'Node'. */  
/* 2 --> Means that the list 'ListCycle' has been build */  
/* in such a way that there is a cycle in it so we just */  
/* de-stack the calls and do not touch anymore the list */  
/* 'ListCycle' that will be used to print out. */  
/*-----*/  
GNL_STATUS GnlMaybeCycle (Node, Depth, ListCycle, Res)  
{  
    GNL_NODE Node;  
    int Depth;  
    BLIST ListCycle;  
    int *Res;  
}
```

gnlcheck.c

```
int i;
GML_VAR Var;
GML_NODE SonI;
GML_NODE NodeFunction;
int RecRes;
GML_NODE SinkNode;

/* We reach a leaf with depth = MAX_DEPTH_CYCLE ... probably this */
/* is a cycle. */
if (Depth > MAX_DEPTH_CYCLE)
{
    BSize (ListCycle) = 0;
    if (BListAddElt (ListCycle, (int)Node))
        return (GML_MEMORY_FULL);
    *Res = 1;
    return (GML_OK);
}

if (GnlNodeOp (Node) == GML_CONSTANTE)
{
    *Res = 0;
    return (GML_OK);
}

if (GnlNodeOp (Node) == GML_VARIABLE)
{
    Var = (GML_VAR)GnlNodeSons (Node);

    if (GnlVarDir (Var) == GML_VAR_INPUT)
    {
        *Res = 0;
        return (GML_OK);
    }

    /* already visited and we know that there is no loop */
    if (GnlVarHook (Var))
    {
        *Res = 0;
        return (GML_OK);
    }

    if (GnlVarFunction (Var) == NULL)
    {
        *Res = 0;
        return (GML_OK);
    }

    NodeFunction = GnlFunctionOnSet (GnlVarFunction (Var));

    if (GnlMaybeCycle (NodeFunction, Depth+1, ListCycle, &RecRes))
        return (GML_MEMORY_FULL);

    /* a potential cycle already reached and 'ListCycle' already built*/
    if (RecRes == 2)
    {
        *Res = 2;
    }
}
```

gnlcheck.c

```
        return (GNL_OK);
    }

/* a potential cycle already reached but we are currently building*/
/* the 'ListCycle'. */
if (RecRes == 1)
{
    SinkNode = (GNL_NODE)BListElt (ListCycle, 0);
    /* Ok the loop is here at that node. */
    if (Node == SinkNode)
    {
        if (BListAddElt (ListCycle, (int)Node))
            return (GNL_MEMORY_FULL);
        *Res = 2;
        return (GNL_OK);
    }

    /* Otherwise we stack the node 'Node'. */
    if (BListAddElt (ListCycle, (int)Node))
        return (GNL_MEMORY_FULL);
    *Res = 1;
    return (GNL_OK);
}

/* We pass thru this variable 'Var' and we know that there is no */
/* when traversing it. So we mark it and will not pass thru next */
/* time. */
SetGnlVarHook (Var, (int*)1);

*Res = 0;
return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons(Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons(Node), i);
    if (GnlMaybeCycle (SonI, Depth+1, ListCycle, &RecRes))
        return (GNL_MEMORY_FULL);

    if (RecRes == 2)
    {
        *Res = 2;
        return (GNL_OK);
    }

    if (RecRes == 1)
    {
        SinkNode = (GNL_NODE)BListElt (ListCycle, 0);
        /* Ok the loop is here at that node. */
        if (Node == SinkNode)
        {
            if (BListAddElt (ListCycle, (int)Node))
                return (GNL_MEMORY_FULL);
            *Res = 2;
            return (GNL_OK);
        }
    }
}
```

gnlcheck.c

```
/* Otherwise we stack the node 'Node'. */  
    if (BListAddElt (ListCycle, (int)Node))  
        return (GNL_MEMORY_FULL);  
    *Res = 1;  
    return (GNL_OK);  
}  
  
}  
  
*Res = 0;  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlPrintCycle */  
/*-----*/  
void GnlPrintCycle (Gnl, ListCycle)  
{  
    GNL          Gnl;  
    BLIST       ListCycle;  
    int          i;  
    GNL_NODE    NodeI;  
    GNL_VAR     Var;  
  
    fprintf (stderr,  
            " ERROR: Combinatorial cycle (size=%d) detected in [%s]\n",  
            BListSize (ListCycle), GnlVarName (Gnl));  
    for (i=BListSize (ListCycle)-1; i>0; i--)  
    {  
        NodeI = (GNL_NODE)BListElt (ListCycle, i);  
        if (GnlNodeOp (NodeI) == GNL_VARIABLE)  
        {  
            Var = (GNL_VAR)GnlNodeSons (NodeI);  
            fprintf (stderr, "%s ->\n", GnlVarName (Var));  
        }  
        else  
            fprintf (stderr, "... ->\n");  
    }  
  
    NodeI = (GNL_NODE)BListElt (ListCycle, i);  
    if (GnlNodeOp (NodeI) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR)GnlNodeSons (NodeI);  
        fprintf (stderr, "%s\n", GnlVarName (Var));  
    }  
    else  
        fprintf (stderr, "... \n");  
}  
/*-----*/  
/* GnlDetectCyclesInGnl */  
/*-----*/  
/* This procedure detects cycles in a Gnl which is a graph. It uses a */  
/* stack-based approach and is not very efficient. */
```

gnlcheck.c

```
/*-----*/
GNL_STATUS GnlDetectCyclesInGnl (Gnl)
    GNL          Gnl;
{
    int          i;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;
    int          Res;
    BLIST       ListCycle;

    /* Reset the Var hook. */
    GnlResetVarHook (Gnl); */

    if (BListCreate (&ListCycle))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR) BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);

        /* There is a potential cycle. */
        if (GnlMaybeCycle (GnlFunctionOnSet (FunctionI), 0, ListCycle,
                           &Res))
            return (GNL_MEMORY_FULL);

        if (Res == 2)
        {
            GnlPrintCycle (Gnl, ListCycle);
            BListQuickDelete (&ListCycle);
            return (GNL_CYCLE_DETECTED);
        }

        if (Res == 1)
        {
            fprintf (stderr,
                     " ERROR: %s detected a very long path of length > %d\n",
                     GNL_TOOL_NAME,
                     MAX_DEPTH_CYCLE);
            BListQuickDelete (&ListCycle);
            return (GNL_CYCLE_DETECTED);
        }
    }

    BListQuickDelete (&ListCycle);

    /* Reset the Var hook. */
    GnlResetVarHook (Gnl); */

    return (GNL_OK);
}

/*-----*/
/* GnlDetectUnusedVarInGnl */
/*-----*/
```

gnlcheck.c

```
/*-----*/
/* This procedure extracts the variables with a functionality but which */
/* are not used anywhere. */
/*-----*/
GNL_STATUS GnlDetectUnusedVarInGnl (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_VAR     Vari;
    int           Line;

    GnlResetDadsInVar (Gnl);

    /* we compute the number of Dads for Variable. */
    GnlGetNumberDadsFromGnl (Gnl);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR) BListElt (GnlFunctions (Gnl), i);
        if ((GnlVarDads (Vari) == 0) &&
            (GnlVarDir (Vari) == GNL_VAR_LOCAL))
        {
            Line = GnlVarLineNumber (Vari);
            fprintf (stderr,
" WARNING (%d): signal <%s> is never used in module [%s]\n",
                Line, GnlVarName (Vari), GnlName (Gnl));
        }
    }

    return (GNL_OK);
}

/*-----*/
/* GnlCheckUndrivenVar */
/*-----*/
/* This procedure extracts the variable which are GNL_VAR_LOCAL and */
/* which have no functionality associated to it. There are undriven */
/* signals and this can cause important troubles for many traversal */
/* algorithms. */
/*-----*/
static BLIST G_UndrivenVar;
GNL_STATUS GnlCheckUndrivenVar (Node)
    GNL_NODE Node;
{
    int           i;
    GNL_VAR     Var;
    GNL_NODE SonI;
    GNL_NODE NodeFunction;
    unsigned int Key;
    BLIST       NewList;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
```

gnlcheck.c

```
{  
    Var = (GNL_VAR) GnlNodeSons (Node);  
  
    if (GnlVarDir (Var) != GNL_VAR_LOCAL)  
        return (GNL_OK);  
  
    Key = KeyOfName (GnlVarName (Var), BListSize (G_UndrivenVar));  
  
    if (G_UndrivenVar->Adress [Key] == (int) NULL)  
    {  
        if (BListCreateWithSize (1, &NewList))  
            return (GNL_MEMORY_FULL);  
        G_UndrivenVar->Adress [Key] = (int) NewList;  
    }  
    NewList = (BLIST) (G_UndrivenVar->Adress [Key]);  
  
    if (GnlVarFunction (Var) == NULL)  
    {  
        if (!BListMemberOfList (NewList, Var, IntIdentical))  
            if (BlistAddElt (NewList, (int) Var))  
                return (GNL_MEMORY_FULL);  
        return (GNL_OK);  
    }  
  
    NodeFunction = GnlFunctionOnSet (GnlVarFunction (Var));  
    if (NodeFunction == NULL)  
    {  
        if (!BListMemberOfList (NewList, Var, IntIdentical))  
            if (BlistAddElt (NewList, (int) Var))  
                return (GNL_MEMORY_FULL);  
        return (GNL_OK);  
    }  
    return (GNL_OK);  
}  
  
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
{  
    SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);  
    if (GnlCheckUndrivenVar (SonI))  
        return (GNL_MEMORY_FULL);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlDetectUndrivenVarInGnl */  
/*-----*/  
/* This procedure enumerates the variables which are used and which are */  
/* supposed to have an associated function (GNL_VAR_LOCAL) but actually */  
/* they have not. */  
/*-----*/  
GNL_STATUS GnlDetectUndrivenVarInGnl (Gnl)  
{  
    GNL          Gnl;  
    int          i;  
    GNL_VAR    VarI;
```

gnlcheck.c

```
GNL_FUNCTION FunctionI;
BLIST ListI;
BLIST BucketJ;
int j;
int Line;

if (BListCreateWithSize (1000, &G_UndrivenVar))
    return (GNL_MEMORY_FULL);
BSize (G_UndrivenVar) = 1000;

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    if (GnlCheckUndrivenVar (GnlFunctionOnSet (FunctionI)))
        return (GNL_MEMORY_FULL);
}

for (j=0; j<BListSize (G_UndrivenVar); j++)
{
    BucketJ = (BLIST)BListElt (G_UndrivenVar, j);
    if (!BucketJ)
        continue;

    for (i=0; i<BListSize (BucketJ); i++)
    {
        VarI = (GNL_VAR)BListElt (BucketJ, i);
        Line = GnlVarLineNumber (VarI);
        fprintf (stderr,
" WARNING (%d): signal <%s> is used but never driven in module [%s]\n",
                Line, GnlVarName (VarI), GnlName (Gnl));
        SetGnlVarDir (VarI, GNL_VAR_LOCAL_UNDRIVEN);
    }
}

for (i=0; i<BListSize (G_UndrivenVar); i++)
{
    ListI = (BLIST)BListElt (G_UndrivenVar, i);
    if (ListI)
        BListQuickDelete (&ListI);
}
BListQuickDelete (&G_UndrivenVar);

return (GNL_OK);
}

/*-----
/* GnlNodeIsConnected
*/
void GnlNodeIsConnected (Node)
    GNL_NODE Node;
{
    int i;
    GNL_VAR Var;
    GNL_FUNCTION Function;
    GNL_NODE NodeI;
```

gnlcheck.c

```
if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    Var = (GNL_VAR) GnlNodeSons (Node);

    /* already processed. */
    if (GnlVarHook (Var))
        return;

    SetGnlVarHook (Var, 1);      /* We mark the variable. */

    if (!GnlVarFunction (Var))
        return;

    Function = GnlVarFunction (Var);
    Node = GnlFunctionOnSet (Function);
    GnlNodeIsConnected (Node);

    return;
}

if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return;

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    NodeI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
    GnlNodeIsConnected (NodeI);
}
}

/*-----*/
/* GnlRemoveNonConnexNodes */
/*-----*/
GNL_STATUS GnlRemoveNonConnexNodes (Gnl)
    GNL          Gnl;
{
    int          i;
    int          j;
    GNL_VAR     VarJ;
    GNL_VAR     VarI;
    BLIST       BucketI;
    GNL_FUNCTION Function;
    GNL_NODE    Node;
    BLIST       HashTableNames;

    GnlResetVarHook (Gnl);

    HashTableNames = GnlHashNames (Gnl);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST) BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
```

gnlcheck.c

```
VarJ = (GNL_VAR)BListElt (BucketI, j);

/* This var has been already traversed. */
if (GnlVarHook (VarJ))
    continue;

if (GnlVarDir (VarJ) == GNL_VAR_LOCAL)
    fprintf (stderr, "%s\n", GnlVarName (VarJ));

continue;
    if ((GnlVarDir (VarJ) == GNL_VAR_OUTPUT) ||
        (GnlVarDir (VarJ) == GNL_VAR_INOUT) ||
        (GnlVarDir (VarJ) == GNL_VAR_LOCAL_WIRING))
    {
        if (!GnlVarFunction (VarJ))
            continue;

        Function = GnlVarFunction (VarJ);
        Node = GnlFunctionOnSet (Function);
        GnlNodeIsConnected (Node);
    }
}
}

for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        if (!GnlVarHook (VarJ) &&
            (GnlVarDir (VarJ) == GNL_VAR_LOCAL))
        {
            BListDelInsert (BucketI, j+1);
            j--;
        }
    }
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    if (!GnlVarHook (VarI) &&
        (GnlVarDir (VarI) == GNL_VAR_LOCAL))
    {
        fprintf (stderr,
                 " WARNING: Removing unconnected signal '%s'\n",
                 GnlVarName (VarI));

        BListDelInsert (GnlFunctions (Gnl), i+1);
        i--;
    }
}

GnlResetVarHook (Gnl);
```

gnlcheck.c

```
    return (GNL_OK);
}

/*-----
/* GnlCheckGnl
/*-----
/* Three kind of checks are implemented: */
/*   - cycle detection in Boolean equations in a Gnl           */
/*   - warning enumeration of unused variables having a functionality*/
/*   - warning enumeration of undriven variables supposed to have a      */
/*     functionality                                         */
/*-----*/
GNL_STATUS GnlCheckGnl (Gnl)
    GNL          Gnl;
{
    GNL_STATUS    Status;

    /* Cycles detections. */
    if ((Status = GnlDetectCyclesInGnl (Gnl)))
        return (Status);

    /* Unused var detections. */
    if (GnlDetectUnusedVarInGnl (Gnl))
        return (GNL_MEMORY_FULL);

    /* undriven var detections. */
    if (GnlDetectUndrivenVarInGnl (Gnl))
        return (GNL_MEMORY_FULL);

    /*
     if (GnlRemoveNonConnexNodes (Gnl))
         return (GNL_MEMORY_FULL);
    */

    return (GNL_OK);
}

/*-----
/* GnlCheckAllGnl
/*-----*/
GNL_STATUS GnlCheckAllGnl (Nw, Gnl)
    GNL_NETWORK    Nw;
    GNL          Gnl;
{
    int            i;
    GNL_COMPONENT  ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL          GnlCompoI;
    GNL_STATUS    Status;
    BLIST         Components;

    /* If the 'Gnl' has been already processed it is not necessary to      */
    /* re-check it and its user components                                     */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);
```

gnlcheck.c

```
SetGnlTag (Gnl, GnlNetworkTag (Nw)) ;

/* We check the current 'Gnl'. */
if ((Status = GnlCheckGnl (Gnl)))
{
    if (Status == GNL_CYCLE_DETECTED)
    {
        fprintf (stderr,
                  " Mapit Aborted because of a combinatorial cycle\n");
        exit (1);
    }
    return (GNL_MEMORY_FULL);
}

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (!GnlCompoI)
        continue;

    if (GnlCheckAllGnl (Nw, GnlCompoI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlCheckNetwork
 */
/*
 * Main checking function which checks the correct integrity of the
 * network 'Nw'.
 */
GNL_STATUS GnlCheckNetwork (Nw)
    GNL_NETWORK      Nw;
{
    GNL          TopGnl;

    TopGnl = GnlNetworkTopGnl (Nw);
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    if (GnlCheckAllGnl (Nw, TopGnl))
        return (GNL_MEMORY_FULL);

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    return (GNL_OK);
}
```

gnlcheck.c

/*-----*/

© 1997-2002 by the Board of Trustees of the Leland Stanford Junior University. All rights reserved.

gnlcreate.c

```
/*-----*/
/*
 *      File:          gnlcreate.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 *
 *      Description:          */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "tokens.h"        /* Actually the tokens from y.tab.h coming from */
                         /* from the <yacc -d gnlparse.y> */

#include "blist.e"

/*-----*/
/* GLOBAL VARIABLES          */
/*-----*/
int GNL_VAR_UNIQUE_ID = 0; /* Unique Id identifying GNL_VAR */

/*-----*/
/* GNL_CALLOC          */
/*-----*/
unsigned GNL_CALLOC (NbObj, SizeObj)
{
    unsigned NbObj;
    unsigned SizeObj;
}

#ifdef MEMORY_STAT
    GLOBAL->TotalMemoryAlloc += NbObj*SizeObj;
#endif

    return ((unsigned)malloc (NbObj, SizeObj));
}

/*-----*/
/* KeyOfName          */
/*-----*/
unsigned int KeyOfName (Name, HashTableSize)
{
    char      *Name;
    int       HashTableSize;
}

    char      *p;
    unsigned   h;
    unsigned   g;
```

gnlcreate.c

```
h = 0;
for (p= Name; *p ;p++)
{
    h= (h<<4)+(*p);
    if ((g = h & 0xF0000000) != 0)
    {
        h ^= (g>>24);
        h ^= g;
    }
}

return (h % HashTableSize);
}

/*
/* GnlCreateBufComponent
*/
GNL_STATUS GnlCreateAssoc ();
GNL_STATUS GnlCreateBufComponent (NewCompo)
    GNL_BUF_COMPONENT      *NewCompo;
{
    BLIST      NewList;
    int       i;
    GNL_ASSOC  NewAssoc;

    if ((*NewCompo = (GNL_BUF_COMPONENT)calloc (1,
                                                sizeof (GNL_BUF_COMPONENT_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlBufType (*NewCompo, GNL_BUF_COMPO);

    if (BListCreateWithSize (2, &NewList))
        return (GNL_MEMORY_FULL);

    for (i=0; i<2; i++)
    {
        if (GnlCreateAssoc (&NewAssoc))
            return (GNL_MEMORY_FULL);

        if (BListAddElt (NewList, (int)NewAssoc))
            return (GNL_MEMORY_FULL);
    }

    SetGnlBufInterface (*NewCompo, NewList);

    return (GNL_OK);
}

/*
/* GnlCreateTristateComponent
*/
GNL_STATUS GnlCreateAssoc ();
GNL_STATUS GnlCreateTristateComponent (NewCompo)
    GNL_TRISTATE_COMPONENT      *NewCompo;
```

gnlcreate.c

```
{  
    BLIST      NewList;  
    int         i;  
    GNL_ASSOC   NewAssoc;  
  
    if ((*NewCompo = (GNL_TRISTATE_COMPONENT)malloc (1,  
                                                    sizeof (GNL_TRISTATE_COMPONENT_REC))) == NULL)  
        return (GNL_MEMORY_FULL);  
  
    SetGnlTriStateType (*NewCompo, GNL_TRISTATE_COMPO);  
  
    if (BListCreateWithSize (3, &NewList))  
        return (GNL_MEMORY_FULL);  
  
    for (i=0; i<3; i++)  
    {  
        if (GnlCreateAssoc (&NewAssoc))  
            return (GNL_MEMORY_FULL);  
  
        if (BListAddElt (NewList, (int)NewAssoc))  
            return (GNL_MEMORY_FULL);  
    }  
  
    SetGnlTriStateInterface (*NewCompo, NewList);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlVarCreate */  
/*-----*/  
GNL_STATUS GnlVarCreate (Gnl, Name, Var)  
{  
    GNL           Gnl;  
    char          *Name;  
    GNL_VAR      *Var;  
    char          *NewName;  
  
    if ((*Var = (GNL_VAR)GNL_CALLOC (1, sizeof(GNL_VAR_REC))) == NULL)  
        return (GNL_MEMORY_FULL);  
  
    SetGnlVarDir ((*Var), GNL_VAR_LOCAL); /* By Default */  
    SetGnlVarType ((*Var), GNL_VAR_INTERNAL); /* By Default */  
    SetGnlVarMsb ((*Var), -1); /* By Default */  
    SetGnlVarLsb ((*Var), -1); /* By Default */  
    SetGnlVarId ((*Var), GNL_VAR_UNIQUE_ID++);  
  
    if (GnlStrCopy (Name, &NewName))  
        return (GNL_MEMORY_FULL);  
    SetGnlVarName ((*Var), NewName);  
  
    return (GNL_OK);  
}  
  
/*-----*/
```

gnlcreate.c

```
/* GnlResizeHashNames */  
/*-----*/  
/* This procedure reduce the size of a hash table names if this one is */  
/* to huge compare to the number of elements it stores. */  
/*-----*/  
#define GNL_THRESHOLD_RESIZE 20  
GNL_Status GnlResizeHashNames (Gnl)  
    GNL      Gnl;  
{  
    int          i;  
    int          j;  
    int          NbElem;  
    BLIST      HashTableNames;  
    BLIST      NewList;  
    BLIST      BucketI;  
    GNL_VAR    VarJ;  
  
    NbElem = 0;  
  
    HashTableNames = GnlHashNames (Gnl);  
    for (i=0; i<BListSize (HashTableNames); i++)  
    {  
        BucketI = (BLIST)BListElt (HashTableNames, i);  
        if (BucketI)  
            NbElem += BListSize (BucketI);  
        if (NbElem > GNL_THRESHOLD_RESIZE)  
            return (GNL_OK);  
    }  
  
    if (BListCreateWithSize (NbElem, &NewList))  
        return (GNL_MEMORY_FULL);  
  
    HashTableNames = GnlHashNames (Gnl);  
    for (i=0; i<BListSize (HashTableNames); i++)  
    {  
        BucketI = (BLIST)BListElt (HashTableNames, i);  
        if (!BucketI)  
            continue;  
  
        for (j=0; j<BListSize (BucketI); j++)  
        {  
            VarJ = (GNL_VAR)BListElt (BucketI, j);  
  
            if (BListAddElt (NewList, (int)VarJ))  
                return (GNL_MEMORY_FULL);  
        }  
        BListQuickDelete (&BucketI);  
    }  
  
    BListQuickDelete (&HashTableNames);  
  
    if (BListCreateWithSize (1, &HashTableNames))  
        return (GNL_MEMORY_FULL);  
    if (BListAddElt (HashTableNames, (int)NewList))  
        return (GNL_MEMORY_FULL);
```

gnlcreate.c

```
SetGnlHashNames (Gnl, HashTableNames);

return (GNL_OK);

}

/*-----
/* GnlUpdateDirVar
/*-----*/
/* 'Var' is used in the components and its direction must be updated */
/* (and the direction of its expansions) if its Direction is local */
/* GNL_VAR_LOCAL. We change it into GNL_VAR_LOCAL_WIRING so that it */
/* is still Local but cannot be removed from any optimization phases. */
/*-----*/
GNL_STATUS GnlUpdateDirVar (Gnl, Var)
    GNL          Gnl;
    GNL_VAR    Var;
{
    int           i;
    int           Start;
    int           End;
    char          *IndexVarName;
    GNL_VAR  IndexVar;

    if (GnlVarRangeUndefined (Var))      /* it is a single bit      */
    {
        if (GnlVarDir (Var) == GNL_VAR_LOCAL)
            SetGnlVarDir (Var, GNL_VAR_LOCAL_WIRING);

        return (GNL_OK);
    }

    /* If not it is a bus signal. */
    if (GnlVarMsb (Var) >= GnlVarLsb (Var))
    {
        Start = GnlVarMsb (Var);
        End = GnlVarLsb (Var);
    }
    else
    {
        Start = GnlVarLsb (Var);
        End = GnlVarMsb (Var);
    }

    for (i = Start; i >= End; i--)
    {
        if (GnlVarIndexName (Var, i, &IndexVarName))
            return (GNL_MEMORY_FULL);

        if (GnlGetVarFromName (Gnl, IndexVarName, &IndexVar))
            GnlError (12 /* Var does not exists */);

        if (GnlVarDir (IndexVar) == GNL_VAR_LOCAL)
            SetGnlVarDir (IndexVar, GNL_VAR_LOCAL_WIRING);
    }
}
```

gnlcreate.c

```
        free ((char*)IndexVarName);
    }

    return (GNL_OK);
}

/*
 * GnlUpdateDir
 */
/*
 * 'Var' is used in the components and its direction must be updated
 * (and the direction of its expansions) if its Direction is local
 * GNL_VAR_LOCAL. We change it into GNL_VAR_LOCAL_WIRING so that it
 * is still Local but cannot be removed from any optimization phases.
 * 'Var' can be also a GNL_NODE with operator GNL_CONCAT.
 */
GNL_STATUS GnlUpdateDir (Gnl, Var)
    GNL          Gnl;
    GNL_VAR     Var;
{
    int           i;
    GNL_VAR   Vari;
    BLIST      Sons;

    if (!Var)
        return (GNL_OK);

    if (GnlVarIsVar (Var))
        return (GnlUpdateDirVar (Gnl, Var));

    Sons = GnlNodeSons ((GNL_NODE)Var);
    for (i=0; i<BListSize (Sons); i++)
    {
        Vari = (GNL_VAR)BListElt (Sons, i);
        if (GnlUpdateDirVar (Gnl, Vari))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 * GnlUniqueVarNameInGnl
 */
/*
 * This procedure returns 1 if the name 'Name' is unique and 0 if the
 * Name already exists.
 */
int GnlUniqueVarNameInGnl (Gnl, Name)
    GNL          Gnl;
    char         *Name;
{
    BLIST      NewList;
    GNL_VAR   Vari;
    unsigned int  Key;
    int           i;
```

gnlcreate.c

```
BLIST      HashTableNames;
GNL_STATUS   GnlStatus;
char        *NewName;

HashTableNames = GnlHashNames (Gnl);

Key = KeyOfName (Name, BListSize (HashTableNames));

if (HashTableNames->Adress [Key] == (int)NULL)
{
    return (1);
}

NewList = (BLIST) (HashTableNames->Adress [Key]);

for (i=0; i<BListSize (NewList); i++)
{
    VarI = (GNL_VAR)BListElt (NewList, i);
    if (!strcmp (GnlVarName (VarI), Name))
    {
        return (0);
    }
}

return (1);
}

/*
 * GnlVarCreateAndAddInHashTableWithNoTest
 */
/*
 * Create a new var without testing if the var already exists.
 */
/*
 * This procedure should then be used very carefully. Give a name that
 */
/*
 * you know it will be unique.
 */
/*
 * GNL_STATUS GnlVarCreateAndAddInHashTableWithNoTest (Gnl, Name, Var)
 */
GNL          Gnl;
char        *Name;
GNL_VAR    *Var;
{
    BLIST      NewList;
    GNL_VAR   VarI;
    unsigned int Key;
    int         i;
    BLIST      HashTableNames;
    GNL_STATUS   GnlStatus;
    char        *NewName;

HashTableNames = GnlHashNames (Gnl);

Key = KeyOfName (Name, BListSize (HashTableNames));

if (HashTableNames->Adress [Key] == (int)NULL)
{
    if (BListCreateWithSize (1, &NewList))
```

gnlcreate.c

```
        return (GNL_MEMORY_FULL);

        HashTableNames->Adress[Key] = (int)NewList;
    }

NewList = (BLIST)(HashTableNames->Adress[Key]);

.. if (GnlStrCopy (Name, &NewName))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlVarCreate (Gnl, NewName, Var)))
    return (GnlStatus);

if (BListAddElt ((BLIST)(HashTableNames->Adress[Key]), (int)(*Var)))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* GnlVarCreateAndAddInHashTable
/*-----*/
GNL_STATUS GnlVarCreateAndAddInHashTable (Gnl, Name, Var)
    GNL          Gnl;
    char         *Name;
    GNL_VAR     *Var;
{
    BLIST      NewList;
    GNL_VAR   VarI;
    unsigned int Key;
    int         i;
    BLIST      HashTableNames;
    GNL_STATUS  GnlStatus;
    char         *NewName;

HashTableNames = GnlHashNames (Gnl);

/* The list GnlHashNames (Gnl) is of size HASH_TABLE_NAMES_SIZE */
Key = KeyOfName (Name, BListSize (HashTableNames));

if (HashTableNames->Adress[Key] == (int)NULL)
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);

    HashTableNames->Adress[Key] = (int)NewList;
}

NewList = (BLIST)(HashTableNames->Adress[Key]);

for (i=0; i<BListSize (NewList); i++)
{
    VarI = (GNL_VAR)BListElt (NewList, i);
```

gnlcreate.c

```
if (!strcmp (GnlVarName (VarI), Name))
{
    *Var = VarI;
    return (GNL_VAR_EXISTS);
}

/* If not found we create physically a new GNL_VAR */
if (GnlStrCopy (Name, &NewName))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlVarCreate (Gnl, NewName, Var)) )
    return (GnlStatus);

if (BListAddElt ((BLIST) (HashTableNames->Adress [Key]), (int) (*Var)))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}
```

```
/*
 *-----*/
/* GnlFreeGnlNode */
/*-----*/
/* This procedure frees virtually the GNL_NODE 'Node' and stores it in */
/* a free list. Never free physically a GNL_NODE but pass thru this */
/* procedure. */
/*-----*/
GNL_STATUS GnlFreeGnlNode (Gnl, Node)
    GNL          Gnl;
    GNL_NODE     Node;
{
    BLIST      NewList;

    if ((GnlNodeOp (Node) != GNL_CONSTANTE) &&
        (GnlNodeOp (Node) != GNL_VARIABLE))
        BListQuickDelete (&GnlNodeSons (Node));
    BListQuickDelete (&GnlNodeDads (Node));

    SetGnlNodeTag (Node, 0);
    SetGnlNodeOp (Node, 0);
    SetGnlNodeSons (Node, NULL);
    SetGnlNodeDads (Node, NULL);
    SetGnlNodeLineNumber (Node, 0);
    SetGnlNodeHook (Node, NULL);

    if (!GnlListFreeNodes (Gnl))
    {
        if (BListCreate (&NewList))
            return (GNL_MEMORY_FULL);
        SetGnlListFreeNodes (Gnl, NewList);
    }

    if (BListAddElt (GnlListFreeNodes (Gnl), (int)Node))
        return (GNL_MEMORY_FULL);
```

gnlcreate.c

```
    return (GNL_OK);
}

/*
 * GnlCreateNode
 */
/*
 * Each created GNL_NODE object is relative to a 'Gnl' and belongs to
 * a segment of GNL_NODE.
 */
/* You cannot free directly a specific GNL_NODE created in this way
 * because then you free the whole segment.
 */
/* To Free virtually a GNL_NODE use the function 'GnlFreeGnlNode'. */
*/
GNL_STATUS GnlCreateNode (Gnl, Op, NewNode)
    GNL          Gnl;
    GNL_OP       Op;
    GNL_NODE    *NewNode;
{
    int         Size;

    /* If there are some previously freed GNL_NODE we pick up these ones */
    /* first. */
    if (GnlListFreeNodes (Gnl))
    {
        if (Size = BListSize (GnlListFreeNodes (Gnl)))
        {
            *NewNode = (GNL_NODE)BListElt (GnlListFreeNodes (Gnl), Size-1);
            BListDelShift (GnlListFreeNodes (Gnl), Size);
            SetGnlNodeOp ((*NewNode), Op);
            return (GNL_OK);
        }
    }

    /* If the first Node equals the last node we create a new page */
    if (GnlFirstNode (Gnl) >= GnlLastNode (Gnl))
    {
        /* We create a new segment of Nodes. */
        if ((*NewNode = (GNL_NODE)
            GNL_CALLOC (SEGMENT_NODE_SIZE, sizeof(GNL_NODE_REC))) == NULL)
            return (GNL_MEMORY_FULL);
        SetGnlFirstNode (Gnl, *NewNode);
        SetGnlLastNode (Gnl, (*NewNode)+SEGMENT_NODE_SIZE);
        if (BListAddElt (GnlNodesSegments (Gnl), (int)(*NewNode)))
            return (GNL_MEMORY_FULL);
    }
    *NewNode = GnlFirstNode (Gnl);
    SetGnlFirstNode (Gnl, (*NewNode)+1);

    SetGnlNodeOp ((*NewNode), Op);

    return (GNL_OK);
}

/*
 * GnlCreateConcatNode
 */
*/
```

gnlcreate.c

```
/* This GNL_NODE is created independently of any Gnl and is leaved alone */
/* so it can bee freed alone. */
```

```
/*-----*/
GNL_STATUS GnlCreateConcatNode (NewNode)
    GNL_NODE      *NewNode;
{
```

```
    if ((*NewNode = (GNL_NODE)
          GNL_CALLOC (1, sizeof(GNL_NODE_REC))) == NULL)
        return (GNL_MEMORY_FULL);
```

```
    SetGnlNodeOp ((*NewNode), GNL_CONCAT);
```

```
    return (GNL_OK);
}
```

```
/*-----*/
/* GnlCreateNodeForVar */
/*-----*/
GNL_STATUS GnlCreateNodeForVar (Gnl, Var, NewNode)
    GNL           Gnl;
    GNL_VAR       Var;
    GNL_NODE     *NewNode;
{
```

```
    if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))
        return (GNL_MEMORY_FULL);
```

```
    SetGnlNodeSons ((*NewNode), (BLIST)Var);
        /* For GNL_VARIABLE the pointer */
        /* sons correspond to Var */
```

```
    return (GNL_OK);
}
```

```
/*-----*/
/* GnlCreateSequentialComponent */
/*-----*/
GNL_STATUS GnlCreateSequentialComponent (Op, NewPBox)
    GNL_SEQUENTIAL_OP      Op;
    GNL_SEQUENTIAL_COMPONENT *NewPBox;
{
```

```
    BLIST      NewList;
    int         i;
    GNL_ASSOC   NewAssoc;
```

```
    if ((*NewPBox = (GNL_SEQUENTIAL_COMPONENT)
          GNL_CALLOC (1, sizeof(GNL_SEQUENTIAL_COMPONENT_REC))) == NULL)
        return (GNL_MEMORY_FULL);
```

```
    SetGnlSequentialCompoType ((*NewPBox), GNL_SEQUENTIAL_COMPO);
    SetGnlSequentialCompoOp ((*NewPBox), Op);
    /* A priori the sequential component has two outputs which are Q and */
    /* Q bar. */
```

```
    SetGnlSequentialCompoFormOutput ((*NewPBox), GNL_Q_QBAR);
```

gnlcreate.c

```
if (BListCreateWithSize (6, &NewList))
    return (GNL_MEMORY_FULL);

for (i=0; i<6; i++)
{
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (NewList, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}

SetGnlSequentialCompoInterface (*NewPBox, NewList);

return (GNL_OK);
}

/*
 * GnlFunctionCreate
 */
GNL_STATUS GnlFunctionCreate (Gnl, LeftVar, RightNode, GnlFunction)
    GNL          Gnl;
    GNL_VAR      LeftVar;
    GNL_NODE     RightNode;
    GNL_FUNCTION *GnlFunction;
{
    GNL_STATUS   GnlStatus;
    GNL_NODE    NewNode;

    if ((*GnlFunction = (GNL_FUNCTION)
        GNL_CALLOC (1, sizeof(GNL_FUNCTION_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlFunctionVar ((*GnlFunction), LeftVar);
    SetGnlFunctionOnSet ((*GnlFunction), RightNode);

    return (GNL_OK);
}

/*
 * GnlFunctionCreateFromVars
 */
/* Creates a Boolean function corresponding to a simple assignment */
/* between two Vars. */
GNL_STATUS GnlFunctionCreateFromVars (Gnl, LeftVar, RightVar,
                                      GnlFunction, LineNumber)
    GNL          Gnl;
    GNL_VAR      LeftVar;
    GNL_VAR      RightVar;
    GNL_FUNCTION *GnlFunction;
    int         LineNumber;
{
```

gnlcreate.c

```
GNL_STATUS      GnlStatus;
GNL_NODE NewNode;

if ((*GnlFunction = (GNL_FUNCTION)
        GNL_CALLOC (1, sizeof(GNL_FUNCTION_REC))) == NULL)
    return (GNL_MEMORY_FULL);

SetGnlFunctionVar ((*GnlFunction), LeftVar);

/* Create the GNL NODE for Var 'RightVar' */
if (GnlCreateNodeForVar (Gnl, RightVar, &NewNode))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (NewNode, LineNumber);

SetGnlFunctionOnSet ((*GnlFunction), NewNode);

SetGnlVarFunction (LeftVar, (*GnlFunction));

if (BListAddElt (GnlFunctions (Gnl), (int)LeftVar))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlSplitSignal
 */
/*
 * Split signal Var according to its range and put it in the lists
 * GnlInputs or GnlOutputs or GnlLocals according to its Dir.
 */
GNL_STATUS GnlSplitSignal (Gnl, Var)
    GNL          Gnl;
    GNL_VAR      Var;
{
    int           i;
    char          *IndexVarName;
    GNL_VAR      NewVar;
    int           Start;
    int           End;

    if (GnlVarRangeUndefined (Var))      /* so it is then a single bit */
    {
        switch (GnlVarDir (Var)) {
            case GNL_VAR_INPUT:
                if (BListAddElt (GnlInputs (Gnl), (int)Var))
                    return (GNL_MEMORY_FULL);
                SetGnlNbIn (Gnl, GnlNbIn (Gnl) + 1);
                break;

            case GNL_VAR_OUTPUT:
                if (BListAddElt (GnlOutputs (Gnl), (int)Var))
                    return (GNL_MEMORY_FULL);
                SetGnlNbOut (Gnl, GnlNbOut (Gnl) + 1);
        }
    }
}
```

```

        break;

    case GNL_VAR_INOUT:
        if (BListAddElt (GnlInputs (Gnl), (int)Var))
            return (GNL_MEMORY_FULL);
        SetGnlNbIn (Gnl, GnlNbIn (Gnl) + 1);
        if (BListAddElt (GnlOutputs (Gnl), (int)Var))
            return (GNL_MEMORY_FULL);
        SetGnlNbOut (Gnl, GnlNbOut (Gnl) + 1);
        break;

    default:
        if (BListAddElt (GnlLocals (Gnl), (int)Var))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl) + 1);
    }

    return (GNL_OK);
}

/* It is a bus signal that we split into single bit signals */
if (GnlVarMsb (Var) >= GnlVarLsb (Var))
{
    Start = GnlVarMsb (Var);
    End = GnlVarLsb (Var);
}
else
{
    Start = GnlVarLsb (Var);
    End = GnlVarMsb (Var);
}

for (i = Start; i >= End; i--)
{
    if (GnlVarIndexName (Var, i, &IndexVarName))
        return (GNL_MEMORY_FULL);

    if (GnlVarCreate (Gnl, IndexVarName, &NewVar))
        return (GNL_MEMORY_FULL);

    SetGnlVarDir (NewVar, GnlVarDir (Var));

/* the location must be assigned somewhere with the real */
/* location of the indexed signal. */
/*
    SetGnlVarLocation (NewVar, GnlVarLocation (Var));
*/
/* This var is the result of a Bus expansion and is not original*/
SetGnlVarType (NewVar, GNL_VAR_EXPANSED);

switch (GnlVarDir (NewVar)) {
    case GNL_VAR_INPUT:
        if (BListAddElt (GnlInputs (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbIn (Gnl, GnlNbIn (Gnl) + 1);
        break;
}

```

```

        case GNL_VAR_OUTPUT:
            if (BListAddElt (GnlOutputs (Gnl), (int)NewVar))
                return (GNL_MEMORY_FULL);
            SetGnlNbOut (Gnl, GnlNbOut (Gnl) + 1);
            break;

        case GNL_VAR_INOUT:
            if (BListAddElt (GnlInputs (Gnl), (int)NewVar))
                return (GNL_MEMORY_FULL);
            SetGnlNbIn (Gnl, GnlNbIn (Gnl) + 1);
            if (BListAddElt (GnlOutputs (Gnl), (int)NewVar))
                return (GNL_MEMORY_FULL);
            SetGnlNbOut (Gnl, GnlNbOut (Gnl) + 1);
            break;

        default:
            if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
                return (GNL_MEMORY_FULL);
            SetGnlNbLocal (Gnl, GnlNbLocal (Gnl) + 1);
    }

}

/* Adding the original local variable. */
if ((GnlVarDir (Var) != GNL_VAR_INPUT) &&
    (GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
    (GnlVarDir (Var) != GNL_VAR_INOUT))
{
    if (BListAddElt (GnlLocals (Gnl), (int)Var))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl) + 1);
}

return (GNL_OK);
}

/*
 * GnlCreateAllSignals
 */
GNL_STATUS GnlCreateAllSignals (Gnl)
    GNL Gnl;
{
    BLIST      HashTableNames;
    int        i;
    int        j;
    GNL_VAR    VarJ;
    GNL_STATUS GnlStatus;
    BLIST      ListI;
    GNL_VAR    Vari;
}

/* First we scan the HashTableNames in order to split bus signals */
HashTableNames = GnlHashNames (Gnl);

```

gnlcreate.c

```
for (i=0; i<BListSize (HashTableNames); i++)
{
    ListI = (BLIST)BListElt (HashTableNames, i);
    if (ListI)
    {
        for (j=0; j<BListSize (ListI); j++)
        {
            VarJ = (GNL_VAR)BListElt (ListI, j);
            if ((GnlStatus = GnlSplitSignal (Gnl, VarJ)))
                return (GnlStatus);
        }
    }
}

/* We scan all the variables added in GnlInputs (Gnl), GnlOutputs,      */
/* and GnlLocals and add them in the HashTableNames (Gnl).          */
/* Actually signals which are not in the HashTableName are the signal */
/* expanded from the bus signals.                                     */
for (i=0; i<BListSize (GnlInputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BlistElt (GnlInputs (Gnl), i);
    if ((GnlStatus = GnlAddVarInHashTablenames (Gnl, VarI)))
    {
        if (GnlStatus != GNL_VAR_EXISTS)
            return (GnlStatus); /* A real error */ */
    }
}

for (i=0; i<BListSize (GnlOutputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlOutputs (Gnl), i);
    if ((GnlStatus = GnlAddVarInHashTablenames (Gnl, VarI)))
    {
        if (GnlStatus != GNL_VAR_EXISTS)
            return (GnlStatus); /* A real error */ */
    }
}

for (i=0; i<BListSize (GnlLocals (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlLocals (Gnl), i);
    if ((GnlStatus = GnlAddVarInHashTablenames (Gnl, VarI)))
    {
        if (GnlStatus != GNL_VAR_EXISTS)
            return (GnlStatus); /* A real error */ */
    }
}
return (GNL_OK);
}

/*-----
/* GnlOpFromGate
/*-----*/
GNL_OP GnlOpFromGate (Gate)
    int      Gate;
{
```

```

switch (Gate) {
    case G_OR: return (GNL_OR);
    case G_AND: return (GNL_AND);
    case G_NOT: return (GNL_NOT);
    case G_NAND: return (GNL_NAND);
    case G_NOR: return (GNL_NOR);
    case G_XOR: return (GNL_XOR);
    case G_XNOR: return (GNL_XNOR);
    case G_MUX: return (GNL_MUX);
    case G_DFF: return (GNL_DFF);
    case G_DFFX: return (GNL_DFFX);
    case G_DFF1: return (GNL_DFF1);
    case G_DFF0: return (GNL_DFF0);
    case G_LATCH: return (GNL_LATCH);
    case G_LATCHX: return (GNL_LATCHX);
    case G_LATCH1: return (GNL_LATCH1);
    case G_LATCH0: return (GNL_LATCH0);

    default:
        GnlError (8 /* unknown gate */);
        return (GNL_UNKNOWN_GATE);
}
}

/*-----*/
/* GnlAnalyzeBufCompo */
/*-----*/
GNL_STATUS GnlAnalyzeBufCompo (Gnl, Gate, InstanceName, AssocList)
    GNL          Gnl;
    int          Gate;
    char         *InstanceName;
    BLIST        AssocList;
{
    GNL_BUF_COMPONENT      NewBox;
    GNL_VAR               OutputVar;
    GNL_VAR               InputVar;

    if (GnlCreateBufComponent (&NewBox))
        return (GNL_MEMORY_FULL);

    OutputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 0));
    GnlUpdateDir (Gnl, OutputVar);

    InputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 1));
    GnlUpdateDir (Gnl, InputVar);

    SetGnlBufInstName (NewBox, InstanceName);
    SetGnlBufInput (NewBox, InputVar);
    SetGnlBufOutput (NewBox, OutputVar);

    if (BListAddElt (GnlComponents (Gnl), (int)NewBox))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

```

gnlcreate.c

```
/*-----*/
/* GnlAnalyzeTriStateCompo */
/*-----*/
GNL_STATUS GnlAnalyzeTriStateCompo (Gnl, Gate, InstanceName, AssocList)
    GNL          Gnl;
    int          Gate;
    char        *InstanceName;
    BLIST        AssocList;
{
    GNL_TRISTATE_COMPONENT      NewBox;
    GNL_VAR          OutputVar;
    GNL_VAR          InputVar;
    GNL_VAR          SelectVar;
    GNL_VAR          SelectPol;

    if (BListSize (AssocList) != 4)
    {
        fprintf (stderr,
        " ERROR: HUBLE Predefined TriState <%s> requires 4 parameters\n",
        InstanceName);
        exit (1);
    }

    if (GnlCreateTristateComponent (&NewBox))
        return (GNL_MEMORY_FULL);

    OutputVar = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 0));
    GnlUpdateDir (Gnl, OutputVar);

    InputVar = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 1));
    GnlUpdateDir (Gnl, InputVar);

    SelectVar = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 2));
    GnlUpdateDir (Gnl, SelectVar);

    SelectPol = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 3));
    GnlUpdateDir (Gnl, SelectPol);

    SetGnlTriStateInstName (NewBox, InstanceName);
    SetGnlTriStateInput (NewBox, InputVar);
    SetGnlTriStateOutput (NewBox, OutputVar);
    SetGnlTriStateSelect (NewBox, SelectVar);

    if (GnlVarIsVss (SelectPol))
        SetGnlTriStateSelectPol (NewBox, 1);
    else
        SetGnlTriStateSelectPol (NewBox, 0);

    if (BListAddElt (GnlComponents (Gnl), (int)NewBox))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

gnlcreate.c

```
/*-----*/
/* GnlAnalyzeCombUnary */
/*-----*/
GNL_STATUS GnlAnalyzeCombUnary (Gnl, Gate, InstanceName, AssocList,
                                LineNumber)
{
    GNL          Gnl;
    int          Gate;
    char         *InstanceName;
    BLIST        AssocList;
    int          LineNumber;
{
    GNL_VAR     InputVar;
    GNL_VAR     OutputVar;
    GNL_NODE   VarNode;
    GNL_NODE   NewNode;
    BLIST       Sons;
    GNL_FUNCTION NewFunction;

    if (BListSize (AssocList) != 2)
    {
        GnlError (5 /* This gate must have a unique argument */ );
        return (GNL_BAD_ARG_NUMBER);
    }

    if (GnlCreateNode (Gnl, GnlOpFromGate (Gate), &NewNode))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (NewNode, LineNumber);

    InputVar = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 1));
    OutputVar = GnlAssocActualPort ((GNL_ASSOC) BListElt (AssocList, 0));

    if (GnlCreateNodeForVar (Gnl, InputVar, &VarNode))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (VarNode, LineNumber);

    if (BListCreateWithSize (1, &Sons))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (Sons, (int) VarNode))
        return (GNL_MEMORY_FULL);

    SetGnlNodeSons (NewNode, Sons);

    if (GnlFunctionCreate (Gnl, OutputVar, NewNode, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (OutputVar, NewFunction);

    if (BListAddElt (GnlFunctions (Gnl), OutputVar))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

gnlcreate.c

```
/*-----*/
/* GnlCreateNodeVdd */
/*-----*/
GNL_STATUS GnlCreateNodeVdd (Gnl, NodeVdd)
    GNL          Gnl;
    GNL_NODE     *NodeVdd;
{
    BLIST        Sons;

    if (GnlCreateNode (Gnl, GNL_CONSTANTE, NodeVdd))
        return (GNL_MEMORY_FULL);

    SetGnlNodeSons (*NodeVdd, (BLIST)1);

    return (GNL_OK);
}

/*-----*/
/* GnlCreateNodeVss */
/*-----*/
GNL_STATUS GnlCreateNodeVss (Gnl, NodeVss)
    GNL          Gnl;
    GNL_NODE     *NodeVss;
{
    BLIST        Sons;

    if (GnlCreateNode (Gnl, GNL_CONSTANTE, NodeVss))
        return (GNL_MEMORY_FULL);

    SetGnlNodeSons (*NodeVss, (BLIST)0);

    return (GNL_OK);
}

/*-----*/
/* GnlCreateNodeNot */
/*-----*/
/* Creates a new Node object which is the Not of the node 'Node'. */
/*-----*/
GNL_STATUS GnlCreateNodeNot (Gnl, Node, NodeNot)
    GNL          Gnl;
    GNL_NODE     Node;
    GNL_NODE     *NodeNot;
{
    BLIST        Sons;

    if (GnlCreateNode (Gnl, GNL_NOT, NodeNot))
        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (1, &Sons))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (Sons, (int)Node))
        return (GNL_MEMORY_FULL);

    SetGnlNodeSons (*NodeNot, Sons);
```

gnlcreate.c

```
    return (GNL_OK);
}

/*-----*/
/* GnlCreateNodeXor */
/*-----*/
/* Creates a new node which corresponds to the Xor function of Var1/Var2*/
/*      ~Var1.Var2+Var1.~Var2 */
/*-----*/
GNL_STATUS GnlCreateNodeXor (Gnl, Var1, Var2, XorNode, LineNumber)
    GNL           Gnl;
    GNL_NODE Var1;
    GNL_NODE Var2;
    GNL_NODE *XorNode;
    int          LineNumber;
{
    GNL_NODE Var3;
    GNL_NODE Var4;
    GNL_NODE NotVar1;
    GNL_NODE NotVar4;
    GNL_NODE And1;
    GNL_NODE And2;
    GNL_NODE Or;
    BLIST   Sons;

    if (GnlNodeCopyRec (Gnl, Var1, &Var3))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (Var3, LineNumber);

    if (GnlNodeCopyRec (Gnl, Var2, &Var4))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (Var4, LineNumber);

    if (GnlCreateNodeNot (Gnl, Var1, &NotVar1))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (NotVar1, LineNumber);

    if (GnlCreateNodeNot (Gnl, Var4, &NotVar4))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (NotVar4, LineNumber);

    /* Building the first AND ... */
    if (GnlCreateNode (Gnl, GNL_AND, &And1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeLineNumber (And1, LineNumber);
    if (BListCreateWithSize (2, &Sons))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (Sons, (int)NotVar1))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (Sons, (int)Var2))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (And1, Sons);
```

```

/* Building the second AND ... */  

if (GnlCreateNode (Gnl, GNL_AND, &And2))  

    return (GNL_MEMORY_FULL);  

SetGnlNodeLineNumber (And2, LineNumber);  

if (BListCreateWithSize (2, &Sons))  

    return (GNL_MEMORY_FULL);  

if (BListAddElt (Sons, (int)Var3))  

    return (GNL_MEMORY_FULL);  

if (BListAddElt (Sons, (int)NotVar4))  

    return (GNL_MEMORY_FULL);  

SetGnlNodeSons (And2, Sons);  
  

/* Building the OR of the ANDs ... */  

if (GnlCreateNode (Gnl, GNL_OR, &Or))  

    return (GNL_MEMORY_FULL);  

SetGnlNodeLineNumber (Or, LineNumber);  

if (BListCreateWithSize (2, &Sons))  

    return (GNL_MEMORY_FULL);  

if (BListAddElt (Sons, (int)And1))  

    return (GNL_MEMORY_FULL);  

if (BListAddElt (Sons, (int)And2))  

    return (GNL_MEMORY_FULL);  

SetGnlNodeSons (Or, Sons);  
  

*XorNode = Or;  
  

return (GNL_OK);
}  
  

*-----*/  

/* GnlAnalyzeComb2arys */  

/*-----*/  

GNL_STATUS GnlAnalyzeComb2arys (Gnl, Gate, InstanceName, AssocList,  

                                LineNumber)  

{
    GNL          Gnl;
    int          Gate;
    char         *InstanceName;
    BLIST        AssocList;
    int          LineNumber;  

    {  

        int          i;  

        GNL_VAR      Var1;
        GNL_VAR      Var2;
        GNL_VAR      Vari;
        GNL_NODE     Var1Node;
        GNL_NODE     Var2Node;
        GNL_NODE     VariNode;
        GNL_NODE     NewNode;
        GNL_NODE     NotNode;
        GNL_NODE     XorNode;
        BLIST        Sons;
        GNL_FUNCTION NewFunction;
        GNL_VAR      OutputVar;
    }
}

```

gnlcreate.c

```
OutputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 0));

switch (GnlOpFromGate (Gate)) {

    case GNL_AND:
    case GNL_OR:
        if (GnlCreateNode (Gnl, GnlOpFromGate (Gate), &NewNode))
            return (GNL_MEMORY_FULL);

        if (BListCreateWithSize (BListSize (AssocList)-1, &Sons))
            return (GNL_MEMORY_FULL);

        for (i=1; i<BListSize (AssocList); i++)
        {
            VarI = GnlAssocActualPort ((GNL_ASSOC)
                BListElt (AssocList, i));
            if (GnlCreateNodeForVar (Gnl, VarI, &VarINode))
                return (GNL_MEMORY_FULL);
            SetGnlNodeLineNumber (VarINode, LineNumber);

            if (BListAddElt (Sons, (int)VarINode))
                return (GNL_MEMORY_FULL);
        }

        SetGnlNodeSons (NewNode, Sons);
        break;

    case GNL_NAND:
        if (GnlCreateNode (Gnl, GnlOpFromGate (Gate), &NewNode))
            return (GNL_MEMORY_FULL);

        if (BListCreateWithSize (BListSize (AssocList)-1, &Sons))
            return (GNL_MEMORY_FULL);

        for (i=1; i<BListSize (AssocList); i++)
        {
            VarI = GnlAssocActualPort ((GNL_ASSOC)
                BListElt (AssocList, i));
            if (GnlCreateNodeForVar (Gnl, VarI, &VarINode))
                return (GNL_MEMORY_FULL);
            SetGnlNodeLineNumber (VarINode, LineNumber);

            if (BListAddElt (Sons, (int)VarINode))
                return (GNL_MEMORY_FULL);
        }

        SetGnlNodeSons (NewNode, Sons);

        if (GnlCreateNodeNot (Gnl, NewNode, &NotNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeLineNumber (NotNode, LineNumber);
        SetGnlNodeOp (NewNode, GNL_AND);
        NewNode = NotNode;
        break;

    case GNL_NOR:
        if (GnlCreateNode (Gnl, GnlOpFromGate (Gate), &NewNode))
```

```

        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (BListSize (AssocList)-1, &Sons))
        return (GNL_MEMORY_FULL);

    for (i=1; i<BListSize (AssocList); i++)
    {
        VarI = GnlAssocActualPort ((GNL_ASSOC)
                                    BListElt (AssocList, i));
        if (GnlCreateNodeForVar (Gnl, VarI, &VarINode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeLineNumber (VarINode, LineNumber);

        if (BListAddElt (Sons, (int)VarINode))
            return (GNL_MEMORY_FULL);
    }

    SetGnlNodeSons (NewNode, Sons);

    if (GnlCreateNodeNot (Gnl, NewNode, &NotNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeLineNumber (NotNode, LineNumber);
    SetGnlNodeOp (NewNode, GNL_OR);
    NewNode = NotNode;
    break;

case GNL_XOR:
    if (BListSize (AssocList) != 3)
    {
        GnlError (7 /* This gate must have two arguments */ );
        return (GNL_BAD_ARG_NUMBER);
    }

    Var1 = GnlAssocActualPort ((GNL_ASSOC)
                               BListElt (AssocList, 1));
    Var2 = GnlAssocActualPort ((GNL_ASSOC)
                               BListElt (AssocList, 2));

    if (GnlCreateNodeForVar (Gnl, Var1, &Var1Node))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (Var1Node, LineNumber);

    if (GnlCreateNodeForVar (Gnl, Var2, &Var2Node))
        return (GNL_MEMORY_FULL);

    SetGnlNodeLineNumber (Var2Node, LineNumber);

    if (GnlCreateNodeXor (Gnl, Var1Node, Var2Node, &XorNode,
                         LineNumber))
        return (GNL_MEMORY_FULL);
    NewNode = XorNode;
    break;

case GNL_XNOR:
    if (BListSize (AssocList) != 3)
    {

```

```

        GnlError (7 /* This gate must have two arguments */ );
        return (GNL_BAD_ARG_NUMBER);
    }
Var1 = GnlAssocActualPort ((GNL_ASSOC)
                           BListElt (AssocList, 1));
Var2 = GnlAssocActualPort ((GNL_ASSOC)
                           BListElt (AssocList, 2));

if (GnlCreateNodeForVar (Gnl, Var1, &Var1Node))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (Var1Node, LineNumber);

if (GnlCreateNodeForVar (Gnl, Var2, &Var2Node))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (Var2Node, LineNumber);

if (GnlCreateNodeXor (Gnl, Var1Node, Var2Node, &XorNode,
                      LineNumber))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (XorNode, LineNumber);

if (GnlCreateNodeNot (Gnl, XorNode, &NotNode))
    return (GNL_MEMORY_FULL);
NewNode = NotNode;
break;
}

SetGnlNodeLineNumber (NewNode, LineNumber);
if (GnlFunctionCreate (Gnl, OutputVar, NewNode, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (OutputVar, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), OutputVar))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlAnalyzeMUX
 */
GNL_STATUS GnlAnalyzeMUX (Gnl, Gate, InstanceName, AssocList, LineNumber)
    GNL          Gnl;
    int          Gate;
    char         *InstanceName;
    BLIST        AssocList;
    int          LineNumber;
{
    GNL_VAR      Sel;
    GNL_VAR      Var1;
    GNL_VAR      Var2;
    GNL_NODE     SelNode;
}

```

gnlcreate.c

```
GNL_NODE      NotSelNode;
GNL_NODE      Var1Node;
GNL_NODE      Var2Node;
GNL_NODE      And1;
GNL_NODE      And2;
GNL_NODE      Or;
BLIST         Sons;
GNL_FUNCTION  NewFunction;
GNL_VAR       OutputVar;

if (BListSize (AssocList) != 4)
{
    GnlError (8 /* This gate must have three arguments */ );
    return (GNL_BAD_ARG_NUMBER);
}

Sel = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 1));
Var1 = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 2));
Var2 = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 3));
OutputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 0));

if (GnlCreateNodeForVar (Gnl, Sel, &SelNode))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (SelNode, LineNumber);

if (GnlCreateNodeForVar (Gnl, Var1, &Var1Node))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (Var1Node, LineNumber);

if (GnlCreateNodeForVar (Gnl, Var2, &Var2Node))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (Var2Node, LineNumber);

if (GnlCreateNodeNot (Gnl, SelNode, &NotSelNode))
    return (GNL_MEMORY_FULL);

SetGnlNodeLineNumber (NotSelNode, LineNumber);

/* Building the first AND ... */
if (GnlCreateNode (Gnl, GNL_AND, &And1))
    return (GNL_MEMORY_FULL);
SetGnlNodeLineNumber (And1, LineNumber);
if (BListCreateWithSize (2, &Sons))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)SelNode))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)Var1Node))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (And1, Sons);

/* Building the second AND ... */
if (GnlCreateNode (Gnl, GNL_AND, &And2))
    return (GNL_MEMORY_FULL);
```

gnlcreate.c

```
SetGnlNodeLineNumber (And2, LineNumber);
if (BListCreateWithSize (2, &Sons))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)NotSelNode))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)Var2Node))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (And2, Sons);

/* Building the OR of the ANDs ...
if (GnlCreateNode (Gnl, GNL_OR, &Or))
    return (GNL_MEMORY_FULL);
SetGnlNodeLineNumber (Or, LineNumber);
if (BListCreateWithSize (2, &Sons))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)And1))
    return (GNL_MEMORY_FULL);
if (BListAddElt (Sons, (int)And2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (Or, Sons);

if (GnlFunctionCreate (Gnl, OutputVar, Or, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (OutputVar, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), OutputVar))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlAnalyzeCombinatorial
 */
GNL_STATUS GnlAnalyzeCombinatorial (Gnl, Gate, InstanceName, AssocList,
                                     LineNumber)
{
    GNL          Gnl;
    int          Gate;
    char        *InstanceName;
    BLIST        AssocList;
    int          LineNumber;
}

switch (Gate) {
    case G_NOT:
        return (GnlAnalyzeCombUnary (Gnl, Gate, InstanceName,
                                     AssocList, LineNumber));

    case G_AND:
    case G_NAND:
    case G_OR:
    case G_NOR:
    case G_XOR:
    case G_XNOR:
```

gnlcreate.c

```
        return (GnlAnalyzeComb2arys (Gnl, Gate, InstanceName,
                                      AssocList, LineNumber));

    case G_MUX:
        return (GnlAnalyzeMUX (Gnl, Gate, InstanceName, AssocList,
                               LineNumber));

    default:
        GnlError (4 /* This combinatorial gate is unknown */);
        return (GNL_UNKNOWN_GATE);
    }

    return (GNL_OK);
}

/*-----
/* GnlCreateClockVar
*/
GNL_STATUS GnlCreateClockVar (Clk, NewClockVar)
    GNL_VAR          Clk;
    GNL_CLOCK_VAR   *NewClockVar;
{
    BLIST      NewList;

    if ((*NewClockVar = (GNL_CLOCK_VAR)malloc (1, sizeof(GNL_CLOCK_VAR_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlClockVarClock (*NewClockVar, Clk);
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);

    SetGnlClockVarComponents (*NewClockVar, NewList);

    return (GNL_OK);
}

/*-----
/* GnlCreateAssoc
*/
GNL_STATUS GnlCreateAssoc (NewAssoc)
    GNL_ASSOC        *NewAssoc;
{
    if ((*NewAssoc = (GNL_ASSOC)malloc (1, sizeof(GNL_ASSOC_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlAnalyzeLatch
*/
GNL_STATUS GnlAnalyzeLatch (Gnl, Gate, InstanceName, AssocList)
```

gnlcreate.c

```
GNL          Gnl;
int          Gate;
char         *InstanceName;
BLIST        AssocList;
{
    GNL_SEQUENTIAL_COMPONENT  NewBox;
    GNL_VAR                  OutputVar;
    GNL_VAR                  OutputVarBar;
    GNL_VAR                  InputVar;
    GNL_VAR                  ClkVar;
    GNL_VAR                  ClkPol;
    GNL_VAR                  SetVar;
    GNL_VAR                  SetPol;
    GNL_VAR                  ResetVar;
    GNL_VAR                  ResetPol;
    GNL_COMPONENT            NewCompo;
    GNL_CLOCK_VAR            ClockI;
    int                      i;

    if (GnlCreateSequentialComponent (GnlOpFromGate (Gate), &NewBox))
        return (GNL_MEMORY_FULL);

    OutputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 0));
    GnlUpdateDir (Gnl, OutputVar);

    OutputVarBar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 1));
    GnlUpdateDir (Gnl, OutputVarBar);

    InputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 2));
    GnlUpdateDir (Gnl, InputVar);

    ClkVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 3));
    GnlUpdateDir (Gnl, ClkVar);

    ClkPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 4));
    GnlUpdateDir (Gnl, ClkPol);

    /* eventually we add the new clock in 'GnlClocks (Gnl)'. */
    for (i=0; i<BListSize (GnlClocks (Gnl)); i++)
    {
        ClockI = (GNL_CLOCK_VAR)BListElt (GnlClocks (Gnl), i);
        if (GnlClockVarClock (ClockI) == ClkVar)
            break;
    }
    if (i == BListSize (GnlClocks (Gnl))) /* It is a new clock */
    {
        if (GnlCreateClockVar (ClkVar, &ClockI))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlClocks (Gnl), (int)ClockI))
            return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (GnlClockVarComponents (ClockI), (int)NewBox))
        return (GNL_MEMORY_FULL);

    SetVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 5));
    GnlUpdateDir (Gnl, SetVar);
```

gnlcreate.c

```
SetPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 6));
GnlUpdateDir (Gnl, SetPol);

ResetVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 7));
GnlUpdateDir (Gnl, ResetVar);

ResetPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 8));
GnlUpdateDir (Gnl, ResetPol);

SetGnlSequentialCompoInstName (NewBox, InstanceName);
SetGnlSequentialCompoInput (NewBox, InputVar);

if (GnlVarIsVss (OutputVar))
    SetGnlSequentialCompoOutput (NewBox, NULL);
else
    SetGnlSequentialCompoOutput (NewBox, OutputVar);

if (GnlVarIsVss (OutputVarBar))
    SetGnlSequentialCompoOutputBar (NewBox, NULL);
else
    SetGnlSequentialCompoOutputBar (NewBox, OutputVarBar);

/* clock
SetGnlSequentialCompoClock (NewBox, ClkVar);
if (GnlVarIsVdd (ClkPol))
    SetGnlSequentialCompoClockPol (NewBox, 1);
else
    SetGnlSequentialCompoClockPol (NewBox, 0);

/* Set
SetGnlSequentialCompoSet (NewBox, SetVar);
if (GnlVarIsVdd (SetPol))
    SetGnlSequentialCompoSetPol (NewBox, 1);
else
    SetGnlSequentialCompoSetPol (NewBox, 0);

/* Reset
SetGnlSequentialCompoReset (NewBox, ResetVar);
if (GnlVarIsVdd (ResetPol))
    SetGnlSequentialCompoResetPol (NewBox, 1);
else
    SetGnlSequentialCompoResetPol (NewBox, 0);

if (BListAddElt (GnlComponents (Gnl), (int)NewBox))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlAnalyzeDFF
 */
GNL_STATUS GnlAnalyzeDFF (Gnl, Gate, InstanceName, AssocList)
    GNL        Gnl;
    int        Gate;
    char      *InstanceName;
```

gnlcreate.c

```
BLIST          AssocList;
{
    GNL_SEQUENTIAL_COMPONENT  NewBox;
    GNL_VAR                  OutputVar;
    GNL_VAR                  OutputVarBar;
    GNL_VAR                  InputVar;
    GNL_VAR                  ClkVar;
    GNL_VAR                  ClkPol;
    GNL_VAR                  SetVar;
    GNL_VAR                  ResetVar;
    GNL_COMPONENT            NewCompo;
    int                      i;
    GNL_CLOCK_VAR            ClockI;
    GNL_VAR                  SetPol;
    GNL_VAR                  ResetPol;

    if (GnlCreateSequentialComponent (GnlOpFromGate (Gate), &NewBox))
        return (GNL_MEMORY_FULL);

    OutputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 0));
    GnlUpdateDir (Gnl, OutputVar);

    OutputVarBar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 1));
    GnlUpdateDir (Gnl, OutputVarBar);

    InputVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 2));
    GnlUpdateDir (Gnl, InputVar);

    ClkVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 3));
    GnlUpdateDir (Gnl, ClkVar);

    ClkPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 4));
    GnlUpdateDir (Gnl, ClkPol);

    /* eventually we add the new clock in 'GnlClocks (Gnl)'. */
    for (i=0; i<BListSize (GnlClocks (Gnl)); i++)
    {
        ClockI = (GNL_CLOCK_VAR)BListElt (GnlClocks (Gnl), i);
        if (GnlClockVarClock (ClockI) == ClkVar)
            break;
    }
    if (i == BListSize (GnlClocks (Gnl))) /* It is a new clock */
    {
        if (GnlCreateClockVar (ClkVar, &ClockI))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlClocks (Gnl), (int)ClockI))
            return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (GnlClockVarComponents(ClockI), (int)NewBox))
        return (GNL_MEMORY_FULL);

    SetVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 5));
    GnlUpdateDir (Gnl, SetVar);

    SetPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 6));
    GnlUpdateDir (Gnl, SetPol);
```

```

ResetVar = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 7));
GnlUpdateDir (Gnl, ResetVar);

ResetPol = GnlAssocActualPort ((GNL_ASSOC)BListElt (AssocList, 8));
GnlUpdateDir (Gnl, ResetPol);

SetGnlSequentialCompoInstName (NewBox, InstanceName);
SetGnlSequentialCompoInput (NewBox, InputVar);
SetGnlSequentialCompoClock (NewBox, ClkVar);

if (GnlVarIsVss (OutputVar))
    SetGnlSequentialCompoOutput (NewBox, NULL);
else if (GnlVarIsVdd (OutputVar))
    SetGnlSequentialCompoOutput (NewBox, NULL);
else
    SetGnlSequentialCompoOutput (NewBox, OutputVar);

if (GnlVarIsVss (OutputVarBar))
    SetGnlSequentialCompoOutputBar (NewBox, NULL);
else if (GnlVarIsVss (OutputVarBar))
    SetGnlSequentialCompoOutputBar (NewBox, NULL);
else
    SetGnlSequentialCompoOutputBar (NewBox, OutputVarBar);

/* clock */
if (GnlVarIsVss (ClkPol))
    SetGnlSequentialCompoClockPol (NewBox, 0);
else
    SetGnlSequentialCompoClockPol (NewBox, 1);

/* Set */
SetGnlSequentialCompoSet (NewBox, SetVar);
if (GnlVarIsVss (SetPol))
    SetGnlSequentialCompoSetPol (NewBox, 0);
else
    SetGnlSequentialCompoSetPol (NewBox, 1);

/* Reset */
SetGnlSequentialCompoReset (NewBox, ResetVar);
if (GnlVarIsVss (ResetPol))
    SetGnlSequentialCompoResetPol (NewBox, 0);
else
    SetGnlSequentialCompoResetPol (NewBox, 1);

if (BListAddElt (GnlComponents (Gnl), (int)NewBox))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlAnalyzeSequentialCompo
 */
GNL_STATUS GnlAnalyzeSequentialCompo (Gnl, Gate, InstanceName, AssocList)
    Gnl;

```

gnlcreate.c

```
int          Gate;
char        *InstanceName;
BLIST       AssocList;
{

switch (Gate) {
    case G_LATCH:
    case G_LATCHX:
    case G_LATCH1:
    case G_LATCH0:
        if (BListSize (AssocList) != 9)
        {
            fprintf (stderr,
" ERROR: HUBBLE Predefined Latch <%s> requires 9 parameters\n",
                InstanceName);
            exit (1);
        }
        return (GnlAnalyzeLatch (Gnl, Gate, InstanceName, AssocList));

    case G_DFF:
    case G_DFFX:
    case G_DFF0:
    case G_DFF1:
        if (BListSize (AssocList) != 9)
        {
            fprintf (stderr,
" ERROR: HUBBLE Predefined Dff <%s> requires 9 parameters\n",
                InstanceName);
            exit (1);
        }
        return (GnlAnalyzeDFF (Gnl, Gate, InstanceName, AssocList));
    }
}

/*-----
/* GnlCreateUserComponent
/*-----
/* Remark: we do not use the list 'Parameters' for now.
/*-----*/
GNL_STATUS GnlCreateUserComponent (Name, InstanceName, Parameters,
                                  Interface, NewUserCompo)
{
    char          *Name;
    char          *InstanceName;
    BLIST         Parameters;
    BLIST         Interface;
    GNL_USER_COMPONENT  *NewUserCompo;

    if ((*NewUserCompo = (GNL_USER_COMPONENT)
          calloc (1, sizeof(GNL_USER_COMPONENT_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlUserComponentType ((*NewUserCompo), GNL_USER_COMPO);
    SetGnlUserComponentName ((*NewUserCompo), Name);
```

gnlcreate.c

```
SetGnlUserComponentInstName ((*NewUserCompo), InstanceName);
SetGnlUserComponentInterface ((*NewUserCompo), Interface);

/* During parsing, formal ports are represented by char * types.      */
SetGnlUserComponentFormalType ((*NewUserCompo), GNL_FORMAL_CHAR);

return (GNL_OK);
}

/*
 * GnlAnalyzeUserCompo
 */
GNL_STATUS GnlAnalyzeUserCompo (Gnl, GateName, InstanceName, DefParamList,
                                AssocList, LineNumber)
{
    GNL          Gnl;
    char         *GateName;
    char         *InstanceName;
    BLIST        DefParamList;
    BLIST        AssocList;
    int          LineNumber;

    int          i;
    GNL_VAR      PortVar;
    GNL_USER_COMPONENT UserComponent;
    GNL_COMPONENT NewCompo;

    if (GnlAddCompoNameInHashTablenames (Gnl, &GateName))
        return (GNL_MEMORY_FULL);

    /* Making local signals as local wire signals                         */
    for (i=0; i<BListSize (AssocList); i++)
    {
        PortVar = GnlAssocActualPort ((GNL_ASSOC)
                                      BListElt (AssocList, i));
        GnlUpdateDir (Gnl, PortVar);
    }

    if (GnlCreateUserComponent (GateName, InstanceName, DefParamList,
                               AssocList, &UserComponent))
        return (GNL_MEMORY_FULL);
    SetGnlUserComponentLineNumber (UserComponent, LineNumber);

    if (BListAddElt (GnlComponents (Gnl), (int)UserComponent))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
 * GnlAnalyzeMacroCompo
 */
GNL_STATUS GnlAnalyzeMacroCompo (Gnl, GateName, InstanceName,
                                 DefParamList, AssocList)
{
    GNL          Gnl;
```

gnlcreate.c

```
char *GateName;
char *InstanceName;
BLIST DefParamList;
BLIST AssocList;
{
    int i;
    GNL_VAR PortVar;

    for (i=0; i<BListSize (AssocList); i++)
    {
        PortVar = GnlAssocActualPort ((GNL_ASSOC)
                                       BListElt (AssocList, i));
        GnlUpdateDir (Gnl, PortVar);
    }

    return (GNL_OK);
}

/*-----
/* GnlAnalyzeInstance
*/
/*-----*/
/* This function analyzes a Verilog instance during the parsing and */
/* according to the type of the gate fills up the GNL structure. If the */
/* gate is combinatorial then it fills up GnlFuntions (Gn) otherwise the */
/* GnlComponents (Gnl) for sequential or exotic elements. */
*/
/*-----*/
GNL_STATUS GnlAnalyzeInstance (Gnl, Gate, GateName, InstanceName,
                               DefParamList, AssocList, LineNumber, Mode)
{
    GNL Gnl;
    int Gate;
    char *GateName;
    char *InstanceName;
    BLIST DefParamList;
    BLIST AssocList;
    int LineNumber;
    int Mode; /* If 1 then output is the last element */
              /* of list 'AssocList', otherwise 1rst */
    GNL_ASSOC LastAssoc;

    switch (Gate) {
        case G_AND:
        case G_OR:
        case G_NAND:
        case G_NOT:
        case G_NOR:
        case G_XOR:
        case G_XNOR:
        case G_MUX:
            /* If Mode is 1 then the output is the last element */
            if (Mode)
            {
                LastAssoc = (GNL_ASSOC) BListElt (AssocList,
                                                 BListSize (AssocList)-1);
                BListDelShift (AssocList, BListSize (AssocList));
                if (BListInsertInList (AssocList, LastAssoc, 1))

```

gnlcreate.c

```
        return (GNL_MEMORY_FULL);
    }
    return (GnlAnalyzeCombinatorial (Gnl,
                                    Gate, InstanceName, AssocList,
                                    LineNumber));

case G_DFF:
case G_DFFX:
case G_DFF0:
case G_DFF1:
case G_LATCH:
case G_LATCHX:
case G_LATCH1:
case G_LATCH0:
    return (GnlAnalyzeSequentialCompo (Gnl,
                                    Gate, InstanceName, AssocList));

case G_TRISTATE:
    return (GnlAnalyzeTriStateCompo (Gnl,
                                    Gate, InstanceName, AssocList));

case G_BUF:
    return (GnlAnalyzeBufCompo (Gnl,
                                Gate, InstanceName, AssocList));

case G_PADD:
case G_PCOMP:
    return (GnlAnalyzeMacroCompo (Gnl, Gate, InstanceName,
                                DefParamList, AssocList));

case G_USER_GATE:
    return (GnlAnalyzeUserCompo (Gnl, GateName, InstanceName,
                                DefParamList, AssocList,
                                LineNumber));

default:
    GnlError (4 /* This gate is unknown */);
    return (GNL_UNKNOWN_GATE);

}

return (GNL_OK);
}

/*-----
/* GnlSmallCreate
/*-----
/* Create a basic GNL and set the Name field.
/*-----*/
GNL_STATUS GnlSmallCreate (Name, Gnl)
    char *Name;
    GNL *Gnl;
{
    BLIST      NewList;
    char      *CopyName;
```

gnlcreate.c

```
if ((*Gnl = (GNL) GNL_CALLOC (1, sizeof(GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

if (Name)
{
    if (GnlStrCopy (Name, &CopyName))
        return (GNL_MEMORY_FULL);

    SetGnlName ((*Gnl), CopyName);
}

/* Creating lists for different fields. */
if (BListCreateWithSize (SMALL_HASH_TABLE_NAMES_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlHashNames ((*Gnl), NewList);
BSize (NewList) = SMALL_HASH_TABLE_NAMES_SIZE;

if (BListCreateWithSize (HASH_TABLE_COMPO_NAMES_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlHashCompoNames ((*Gnl), NewList);
BSize (NewList) = HASH_TABLE_COMPO_NAMES_SIZE;

/* Creating List of nodes segments */
if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments ((*Gnl), NewList);
SetGnlFirstNode ((*Gnl), NULL);
SetGnlLastNode ((*Gnl), NULL);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlInputs ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlOutputs ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlLocals ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlFunctions ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlClocks ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlComponents ((*Gnl), NewList);

/* we create a hash list of list of GNL_PATH_COMPONENT */
if (BListCreateWithSize (HASH_LIST_PATH_COMPO_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
BSize (NewList) = HASH_LIST_PATH_COMPO_SIZE;
```

gnlcreate.c

```
SetGnlListPathComponent ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlListSourceFiles ((*Gnl), NewList);

return (GNL_OK);
}

/*-----*/
/* GnlCreate */
/*-----*/
/* Create a basic GNL and set the Name field. */
/*-----*/
GNL_STATUS GnlCreate (Name, Gnl)
    char      *Name;
    GNL       *Gnl;
{
    BLIST     NewList;
    char      *CopyName;

if ((*Gnl = (GNL) GNL_CALLOC (1, sizeof (GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

if (Name)
{
    if (GnlStrCopy (Name, &CopyName))
        return (GNL_MEMORY_FULL);

    SetGnlName ((*Gnl), CopyName);
}

/* Creating lists for different fields. */
if (BListCreateWithSize (HASH_TABLE_NAMES_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlHashNames ((*Gnl), NewList);
BSize (NewList) = HASH_TABLE_NAMES_SIZE;

if (BListCreateWithSize (HASH_TABLE_COMPO_NAMES_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlHashCompoNames ((*Gnl), NewList);
BSize (NewList) = HASH_TABLE_COMPO_NAMES_SIZE;

/* Creating List of nodes segments */
if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments ((*Gnl), NewList);
SetGnlFirstNode ((*Gnl), NULL);
SetGnlLastNode ((*Gnl), NULL);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlInputs ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
```

gnlcreate.c

```
SetGnlOutputs((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlLocals((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlFunctions((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlClocks((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlComponents ((*Gnl), NewList);

/* we create a hash list of list of GNL_PATH_COMPONENT           */
if (BListCreateWithSize (HASH_LIST_PATH_COMPO_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
BSize (NewList) = HASH_LIST_PATH_COMPO_SIZE;
SetGnlListPathComponent ((*Gnl), NewList);

if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlListSourceFiles ((*Gnl), NewList);

return (GNL_OK);
}

/*----- EOF -----*/
```

gnlcube.c

```

/*
 *-----*/
/*      File:          gnlcube.c
 *      Version:        1.1
 *      Modifications: -
 *      Documentation: -
 */
/*      Description:          */
/*-----*/
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"

#include "blist.e"

/*
 *-----*/
GNL_STATUS GnlCubesFromGnlNode ();
GNL_STATUS GnlMakeFunctionFlattenable ();

/*
 *-----*/
/* GLOBAL VARIABLES          */
/*-----*/
int    G_Rank;

/*
 *-----*/
/* GnlCubeStructFree          */
/*-----*/
void GnlCubeStructFree (Cb)
    GNL_CUBE Cb;
{
    free ((char*) GnlCubeHigh (Cb));
    SetGnlCubeHigh (Cb, NULL);
    free ((char*) GnlCubeLow (Cb));
    SetGnlCubeLow (Cb, NULL);
}

/*
 *-----*/
/* GnlCubeCreate          */
/*-----*/
GNL_STATUS GnlCubeCreate (NewCube)
    GNL_CUBE     *NewCube;
{
    if ((*NewCube = (GNL_CUBE) GNL_CALLOC (1, sizeof(GNL_CUBE_REC))) ==
        NULL)
        return (GNL_MEMORY_FULL);
}

```

gnlcube.c

```
    return (GNL_OK);
}

/*-----
/* GnlCubeStructCreate                               */
/*-----*/
GNL_STATUS GnlCubeStructCreate (NbCells, Cube)
    int          NbCells;
    GNL_CUBE   Cube;
{
    unsigned *CellHigh;
    unsigned *CellLow;

    if (MintCreate (NbCells, &CellHigh))
        return (GNL_MEMORY_FULL);

    if (MintCreate (NbCells, &CellLow))
        return (GNL_MEMORY_FULL);

    SetGnlCubeHigh (Cube, CellHigh);
    SetGnlCubeLow (Cube, CellLow);

    return (GNL_OK);
}

/*-----
/* GnlCubeFullCreate                                */
/*-----*/
GNL_STATUS GnlCubeFullCreate (NbCells, NewCube)
    int          NbCells;
    GNL_CUBE   *NewCube;
{
    if (GnlCubeCreate (NewCube))
        return (GNL_MEMORY_FULL);

    if (GnlCubeStructCreate (NbCells, *NewCube))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlCubeAndStructCreateInit                      */
/*-----*/
GNL_STATUS GnlCubeAndStructCreateInit (Cube, NbCells)
    GNL_CUBE   Cube;
    int          NbCells;
{
    if (GnlCubeStructCreate (NbCells, Cube))
        return (GNL_MEMORY_FULL);

    MintInitWithValue (GnlCubeHigh (Cube), NbCells, 0);
    MintInitWithValue (GnlCubeLow (Cube), NbCells, ~0);
```

gnlcube.c

```
    return (GNL_OK);
}

/*-----
/* GnlCubeFullAndCreate
*/
/*-----*/
GNL_STATUS GnlCubeFullAndCreate (NbCells, NewCube)
    int          NbCells;
    GNL_CUBE *NewCube;
{

    if (GnlCubeFullCreate (NbCells, NewCube))
        return (GNL_MEMORY_FULL);

    MintInitWithValue (GnlCubeHigh (*NewCube), NbCells, 0);
    MintInitWithValue (GnlCubeLow (*NewCube), NbCells, ~0);

    return (GNL_OK);
}

/*-----
/* GnlCubeAndExtend
*/
/*-----*/
GNL_STATUS GnlCubeAndExtend (OldSize, NewSize, Cube, ExtCube)
    int          OldSize;
    int          NewSize;
    GNL_CUBE Cube;
    GNL_CUBE *ExtCube;
{

    if (GnlCubeFullAndCreate (NewSize, ExtCube))
        return (GNL_MEMORY_FULL);

    memcpy (GnlCubeHigh (*ExtCube), GnlCubeHigh (Cube), OldSize << 2);
    memcpy (GnlCubeLow (*ExtCube), GnlCubeLow (Cube), OldSize << 2);

    return (GNL_OK);
}

/*-----
/* GnlCubeCopy
*/
/*-----*/
GNL_STATUS GnlCubeCopy (NbCells, Cube, NewCube)
    int          NbCells;
    GNL_CUBE     Cube;
    GNL_CUBE     *NewCube;
{

    if (GnlCubeFullCreate (NbCells, NewCube))
        return (GNL_MEMORY_FULL);

    while (NbCells--)
    {
        GnlCubeHigh(*NewCube) [NbCells] = GnlCubeHigh(Cube) [NbCells];
        GnlCubeLow(*NewCube) [NbCells] = GnlCubeLow(Cube) [NbCells];
    }
}
```

gnlcube.c

```
    return (GNL_OK);
}

/*-----
/* GnlCopyListOfCubes
*/
GNL_STATUS GnlCopyListOfCubes (ListOfCubes, CopyList, NbCells)
    BLIST      ListOfCubes;
    BLIST      *CopyList;
    int        NbCells;
{
    int          i;
    GNL_CUBE   CubeI;
    GNL_CUBE   NewCube;

    if (BListCreateWithSize (2, CopyList))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListOfCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListOfCubes, i);
        if (GnlCubeCopy (NbCells, CubeI, &NewCube))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (*CopyList, (int)NewCube))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlCubeFree
*/
void GnlCubeFree (Cb)
    GNL_CUBE *Cb;
{
    if (*Cb == NULL)
        return;

    GnlCubeStructFree (*Cb);
    free ((char*) (*Cb));
    *Cb = NULL;
}

/*-----
/* GnlCubeInitWithValue
*/
/*-----*/
/* Initialize GNL_CUBE 'Cube' with the following values according to 'C'*/
/* C          High           Low          */
/* '0'         0              0          */
/* '1'         1              1          */
/* '_'         0              1          */
/* '?'         1              0          */
/*-----*/
```

gnlcube.c

```
void GnlCubeInitWithValue (Cube, C, NbCells)
    GNL_CUBE Cube;
    char          *C;
    int           NbCells;
{
    switch (*C) {
        case '0':
            MintInitWithValue (GnlCubeHigh(Cube), NbCells, 0);
            MintInitWithValue (GnlCubeLow(Cube), NbCells, 0);
            break;

        case '1':
            MintInitWithValue (GnlCubeHigh(Cube), NbCells, ~0);
            MintInitWithValue (GnlCubeLow(Cube), NbCells, ~0);
            break;

        case '-':
            MintInitWithValue (GnlCubeHigh(Cube), NbCells, 0);
            MintInitWithValue (GnlCubeLow(Cube), NbCells, ~0);
            break;

        case '?':
            MintInitWithValue (GnlCubeHigh(Cube), NbCells, ~0);
            MintInitWithValue (GnlCubeLow(Cube), NbCells, 0);
            break;
    }
}

/*-----
/* GnlCubeLiteralExistQuick
*/
int GnlCubeLiteralExistQuick (Cube, LitInfo, t, d, mask)
    GNL_CUBE      Cube;
    int          LitInfo;
    int          t;
    int          d;
    unsigned     mask;
{

    if (LitInfo & 1)
        return (!((mask & (*(GnlCubeHigh(Cube) + d))) ||
                  (mask & (*(GnlCubeLow(Cube) + d)))));

    return ((int) (mask & (*(GnlCubeHigh(Cube) + d)) &
                  (*(GnlCubeLow(Cube) + d))));
}

/*-----
/* GnlCubeLiteralExist
*/
/* Returns 1 if GNL_CUBE 'Cube' contains literal defined by 'LitInfo'.
/* 'LitInfo' is an encoded value that defines both the index of the
/* searched literal and its form: 'LitInfo' div 2 is the index, and
/* 'LitInfo' mod 2 is the form (1 : normal form, 0 : complement form)
*/
int GnlCubeLiteralExist (Cube, LitInfo)
    GNL_CUBE      Cube;
```

gnlcube.c

```
int LitInfo;
{
    int t;
    int d;
    unsigned mask;

    t = (LitInfo >> 1) % BITS_PER_INT;
    d = (LitInfo >> 1) / BITS_PER_INT;
    mask = 1 << t;

    if (LitInfo & 1)
        return (!((mask & (*GnlCubeHigh(Cube) + d))) ||
                (mask & (*GnlCubeLow(Cube) + d))));

    return ((int) (mask & (*GnlCubeHigh(Cube) + d)) &
            (*GnlCubeLow(Cube) + d)));
}

/*-----
/* GnlCubeIsTautology
/*-----*/
int GnlCubeIsTautology (Cube, NbCells)
    GNL_CUBE Cube;
    int NbCells;
{
    while (NbCells--)
    {
        if (~(GnlCubeHigh (Cube) [NbCells] ^ GnlCubeLow(Cube) [NbCells]))
            return (0);
    }

    return (1);
}

/*-----
/* GnlCubeIncluded
/*-----*/
int GnlCubeIncluded (NbCells, Cube1, Cube2)
    int NbCells;
    GNL_CUBE Cube1;
    GNL_CUBE Cube2;
{
    while (NbCells--)
    {
        if ((GnlCubeLow(Cube1) [NbCells] & ~GnlCubeLow(Cube2) [NbCells]) |
            (~GnlCubeHigh(Cube1) [NbCells] & GnlCubeHigh(Cube2) [NbCells]))
            return (0);
    }

    return (1);
}

/*-----
/* GnlCubePrint
/*-----*/
/* High: 1 0 1 0
/* Low : 1 0 0 1
/*-----*/
```

gnlcube.c

```
/*          1  0  ~  - */  
/*-----*/  
void GnlCubePrint (File, NbVar, Cube) /*-----*/  
    FILE      *File;  
    int       NbVar;  
    GNL_CUBE Cube;  
{  
    int      N;  
    int      Nb;  
    int      Mask;  
  
    N = 0;  
    while (NbVar)  
    {  
        NbVar -= (Nb = (NbVar > BITS_PER_INT) ? BITS_PER_INT : NbVar);  
        Mask = 1;  
  
        do  
        {  
            if (GnlCubeHigh(Cube) [N] & Mask)  
                if (GnlCubeLow(Cube) [N] & Mask)  
                    fprintf (File, "1");  
                else  
                    fprintf (File, "-");  
            else if (GnlCubeLow(Cube) [N] & Mask)  
                fprintf (File, "-");  
            else  
                fprintf (File, "0");  
  
            Mask <<= 1;  
        }  
  
        while (--Nb);  
        N++;  
    }  
}  
  
/*-----*/  
/* GnlCubeLitNumber */  
/*-----*/  
/* Returns the number of literals in the Cube 'Cube'. */  
/*-----*/  
int GnlCubeLitNumber (Cube, NbCells)  
    GNL_CUBE Cube;  
    int       NbCells;  
{  
    unsigned  V;  
    int       LitNb;  
  
    LitNb = 0;  
  
    while (NbCells--)  
    {  
        V = ~(GnlCubeHigh(Cube) [NbCells] ^ GnlCubeLow(Cube) [NbCells]);  
        LitNb += MintCardinalAllCells (&V, 1);  
    }
```

gnlcube.c

```
    }

    return (LitNb);
}

/*-----
/* GnlListCubesLitNumber
*/
/*-----*/
/* Returns the number of literals in a list of Cubes .
*/
/*-----*/
int GnlListCubesLitNumber (ListCubes, NbCells)
    BLIST      ListCubes;
    int         NbCells;
{
    int          i;
    int          NbLit;
    GNL_CUBE   CubeI;

    NbLit = 0;
    for (i=0; i<BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        NbLit += GnlCubeLitNumber (CubeI, NbCells);
    }
    return (NbLit);
}

/*-----
/* GnlListCubesPrint
*/
/*-----*/
void GnlListCubesPrint (File, NbVar, ListCubes)
    FILE          *File;
    int           NbVar;
    BLIST        ListCubes;
{
    int          i;
    GNL_CUBE   CubeI;

    for (i=0; i<BListSize (ListCubes)-1; i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        GnlCubePrint (File, NbVar, CubeI);
        fprintf (File, " + ");
    }

    CubeI = (GNL_CUBE)BListElt (ListCubes, i);
    GnlCubePrint (File, NbVar, CubeI);
    fprintf (File, "\n");
}

/*-----
/* GnlCubesFromGnlNodeOR
*/
/*-----*/
GNL_STATUS GnlCubesFromGnlNodeOR (Gnl, NbCells, NbVar, GnlNode, ListCubes)
    GNL        Gnl;
```

gnlcube.c

```
int          NbCells;
int          NbVar;
GNL_NODE    GnlNode;
BLIST       *ListCubes;
{
    int          i;
    GNL_NODE  SonI;
    BLIST      Sons;
    BLIST      ListCubesSon;

    if (BListCreate (ListCubes))
        return (GNL_MEMORY_FULL);

    Sons = GnlNodeSons (GnlNode);
    for (i=0; i<BListSize (Sons); i++)
    {
        SonI = (GNL_NODE)BListElt (Sons, i);
        if (GnlCubesFromGnlNode (Gnl, NbCells, NbVar, SonI, &ListCubesSon))
            return (GNL_MEMORY_FULL);

        if (BListSize (ListCubesSon))
        {
            if (BListAppend ((*ListCubes), &ListCubesSon))
                return (GNL_MEMORY_FULL);
            }
        else /* Empty list because Cube is the 0 constante */
            { /* Then it is not necessary to append it. */
                BListQuickDelete (&ListCubesSon);
            }
        }

    return (GNL_OK);
}

/*-----
/* GnlMakeFlattenableNodeOR
/*-----*/
GNL_STATUS GnlMakeFlattenableNodeOR (Gnl, NbCells, NbVar, GnlNode,
                                      ListCubes, MaxCubes, Flattened)
{
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST       *ListCubes;
    int          MaxCubes;
    int          *Flattened;
{
    int          i;
    GNL_NODE  SonI;
    BLIST      Sons;
    BLIST      ListCubesSon;

    if (BListCreate (ListCubes))
        return (GNL_MEMORY_FULL);
```

gnlcube.c

```
Sons = GnlNodeSons (GnlNode);
for (i=0; i<BListSize (Sons); i++)
{
    SonI = (GNL_NODE)BListElt (Sons, i);
    if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar, SonI,
                                    &ListCubesSon, MaxCubes, Flattened))
        return (GNL_MEMORY_FULL);

    if (!(*Flattened))
        return (GNL_OK);

    if (BListSize (ListCubesSon))
    {
        if (BListAppend ((*ListCubes), &ListCubesSon))
            return (GNL_MEMORY_FULL);
    }
    else /* Empty list because Cube is the 0 constante */
    {
        /* Then it is not necessary to append it. */
        BListQuickDelete (&ListCubesSon);
    }

    if (BListSize ((*ListCubes)) > MaxCubes)
    {
        *Flattened = 0;
        return (GNL_OK);
    }
}

*Flattened = 1;
return (GNL_OK);
}

/*
 * GnlCubeCumulativeMult
 */
void GnlCubeCumulativeMult (CumulCube, Cube, NbCells)
    GNL_CUBE CumulCube;
    GNL_CUBE Cube;
    int NbCells;
{
    unsigned *CumulHigh;
    unsigned *CumulLow;
    unsigned *High;
    unsigned *Low;

    CumulHigh = GnlCubeHigh(CumulCube);
    CumulLow = GnlCubeLow(CumulCube);
    High = GnlCubeHigh(Cube);
    Low = GnlCubeLow(Cube);

    while (NbCells--)
    {
        *(CumulHigh++) |= *(High++);
        *(CumulLow++) &= *(Low++);
    }
}
```

gnlcube.c

```
/*-----*/
/* GnlCubeMultiply */
/*-----*/
/* Compute the product of the two cubes 'Cube1' and 'Cube2'. The result */
/* 'CubeMult' can be NULL if the product is 0. */
/*-----*/
GNL_STATUS GnlCubeMultiply (NbCells, Cube1, Cube2, CubeMult)
    int          NbCells;
    GNL_CUBE   Cube1;
    GNL_CUBE   Cube2;
    GNL_CUBE *CubeMult;
{

    if (GnlCubeFullCreate (NbCells, CubeMult))
        return (GNL_MEMORY_FULL);

    while (NbCells--)
    {
        if ((GnlCubeHigh(*CubeMult) [NbCells] = GnlCubeHigh(Cube1) [NbCells] |
            GnlCubeHigh(Cube2) [NbCells])
            &
            ~(GnlCubeLow(*CubeMult) [NbCells] = GnlCubeLow(Cube1) [NbCells] &
            GnlCubeLow(Cube2) [NbCells]))
        {
            GnlCubeFree (CubeMult);
            *CubeMult = NULL;
            return (GNL_OK);
        }
    }

    return (GNL_OK);
}

/*-----*/
/* GnlCubeMultQuickCubeCube */
/*-----*/
/* 'Cube3' wil contain the product of 'Cube1' by 'Cube2' iff the product*/
/* is not NULL */
/*-----*/
void GnlCubeMultQuickCubeCube (Cube1, Cube2, Cube3, NbCells)
    GNL_CUBE   Cube1;
    GNL_CUBE   Cube2;
    GNL_CUBE   Cube3;
    int          NbCells;
{

    while (NbCells--)
    {
        GnlCubeHigh(Cube3) [NbCells] = GnlCubeHigh(Cube1) [NbCells] |
            GnlCubeHigh(Cube2) [NbCells];
        GnlCubeLow(Cube3) [NbCells] = GnlCubeLow(Cube1) [NbCells] &
            GnlCubeLow(Cube2) [NbCells];
    }
}

/*-----*/
```

gnlcube.c

```
/* GnlCubeIdentical */  
/*-----*/  
int GnlCubeIdentical (Cube1, Cube2, NbCells)  
    GNL_CUBE Cube1;  
    GNL_CUBE Cube2;  
    int      NbCells;  
{  
  
    if (!memcmp ((char*)GnlCubeHigh(Cube1), (char*)GnlCubeHigh(Cube2),  
                NbCells << 2))  
        if (!memcmp ((char*)GnlCubeLow(Cube1), (char*)GnlCubeLow(Cube2),  
                    NbCells << 2))  
            return (1);  
        else  
            return (0);  
    else  
        return (0);  
}  
  
/*-----*/  
/* GnlCubeAlreadyInList */  
/*-----*/  
int GnlCubeAlreadyInList (NbCells, Cube, List)  
    int      NbCells;  
    GNL_CUBE Cube;  
    BLIST   List;  
{  
    int      i;  
    GNL_CUBE CubeI;  
  
    return (0);  
  
    for (i=0; i<BListSize (List); i++)  
    {  
        CubeI = (GNL_CUBE)BListElt (List, i);  
        if (GnlCubeIdentical (Cube, CubeI, NbCells))  
            return (1);  
    }  
    return (0);  
}  
  
/*-----*/  
/* GnlCubesListMultiply */  
/*-----*/  
GNL_STATUS GnlCubesListMultiply (NbCells, L1, L2, Lres)  
    int      NbCells;  
    BLIST   L1;  
    BLIST   L2;  
    BLIST   *Lres;  
{  
    int      i;  
    int      j;  
    GNL_CUBE CubeI;  
    GNL_CUBE CubeJ;  
    GNL_CUBE CubeMult;
```

```

if (BListCreate (Lres))
    return (GNL_MEMORY_FULL);

/* If one of the list is 0 then we return an empty list */
if ((BListSize (L1) == 0) || (BListSize (L2) == 0))
    return (GNL_OK);

for (i=0; i<BListSize (L1); i++)
{
    CubeI = (GNL_CUBE)BListElt (L1, i);
    for (j=0; j<BListSize (L2); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (L2, j);
        if (GnlCubeMultiply (NbCells, CubeI, CubeJ, &CubeMult))
            return (GNL_MEMORY_FULL);

        if (CubeMult) /* If NULL then the product was 0 */
        {
            if (GnlCubeAlreadyInList (NbCells, CubeMult, *Lres))
                GnlCubeFree (&CubeMult);
            else
            {
                if (BListAddElt (*Lres, (int)CubeMult))
                    return (GNL_MEMORY_FULL);
            }
        }
    }
}

return (GNL_OK);
}

/*
 * GnlCubesListMultiplyFlattenable
 */
GNL_STATUS GnlCubesListMultiplyFlattenable (NbCells, L1, L2, MaxCubes,
                                             Lres)
{
    int          NbCells;
    BLIST       L1;
    BLIST       L2;
    int          MaxCubes;
    BLIST       *Lres;
{
    int          i;
    int          j;
    GNL_CUBE   CubeI;
    GNL_CUBE   CubeJ;
    GNL_CUBE   CubeMult;

    if (BListCreate (Lres))
        return (GNL_MEMORY_FULL);

    /* If one of the list is 0 then we return an empty list */
    if ((BListSize (L1) == 0) || (BListSize (L2) == 0))

```

gnlcube.c

```
    return (GNL_OK);

for (i=0; i<BListSize (L1); i++)
{
    CubeI = (GNL_CUBE)BListElt (L1, i);
    for (j=0; j<BListSize (L2); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (L2, j);
        if (GnlCubeMultiply (NbCells, CubeI, CubeJ, &CubeMult))
            return (GNL_MEMORY_FULL);

        if (CubeMult) /* If NULL then the product was 0 */
        {
            if (GnlCubeAlreadyInList (NbCells, CubeMult, *Lres))
                GnlCubeFree (&CubeMult);
            else
            {
                if (BListAddElt (*Lres, (int)CubeMult))
                    return (GNL_MEMORY_FULL);
                if (BListSize (*Lres) > MaxCubes)
                    return (GNL_OK);
            }
        }
    }
}

return (GNL_OK);
}

/*-----
/* GnlCubesFromGnlNodeAND
/*-----*/
GNL_STATUS GnlCubesFromGnlNodeAND (Gnl, NbCells, NbVar, GnlNode, ListCubes)
{
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST        *ListCubes;

    BLIST      Sons;
    GNL_NODE  SonI;
    int          i;
    int          NbCubes1;
    int          NbCubes2;
    BLIST      ListCubesSon;
    BLIST      ListRes;
    BLIST      MinimizedListRes;

    Sons = GnlNodeSons (GnlNode);
    SonI = (GNL_NODE)BListElt (Sons, 0);

    if (GnlCubesFromGnlNode (Gnl, NbCells, NbVar, SonI, ListCubes))
        return (GNL_MEMORY_FULL);

    if (BListSize (*ListCubes) == 0) /* Actually the 0 Constante */
        return (GNL_OK);
```

```

for (i=1; i<BListSize (Sons); i++)
{
    SonI = (GNL_NODE)BListElts (Sons, i);
    if (GnlCubesFromGnlNode (Gnl, NbCells, NbVar, SonI, &ListCubesSon))
        return (GNL_MEMORY_FULL);

    NbCubes1 = BListSize (*ListCubes);
    NbCubes2 = BListSize (ListCubesSon);

    if (GnlCubesListMultiply (NbCells, *ListCubes, ListCubesSon,
                               &ListRes))
        return (GNL_MEMORY_FULL);

    /* If the multiplication of the lists of cubes produces a list */
    /* too large (e.g. of size 2 times greater than the sum of the */
    /* two original lists) then we invoke the two-level minimization*/
    if (BListSize (ListRes) > 2 * (NbCubes1 + NbCubes2))
    {
        if (LsMinListCubes (ListRes, NULL, NbVar, &MinimizedListRes))
            return (GNL_MEMORY_FULL);

        BListDelete (&ListRes, GnlCubeFree);
        ListRes = MinimizedListRes;
    }
    BListDelete (ListCubes, GnlCubeFree);
    BListDelete (&ListCubesSon, GnlCubeFree);
    *ListCubes = ListRes;

    if (BListSize (ListRes) == 0)      /* Actually the 0 Constante */
    {
        return (GNL_OK);
    }
}

return (GNL_OK);
}

/*-----
/* GnlMakeFlattenableNodeAND
/*-----*/
GNL_STATUS GnlMakeFlattenableNodeAND (Gnl, NbCells, NbVar, GnlNode,
                                      ListCubes, MaxCubes, Flattened)
{
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST        *ListCubes;
    int          MaxCubes;
    int          *Flattened;
}
BLIST    Sons;
GNL_NODE SonI;
int      i;
int      NbCubes1;
int      NbCubes2;
BLIST   ListCubesSon;

```

gnlcube.c

```
BLIST      ListRes;
BLIST      MinimizedListRes;

Sons = GnlNodeSons (GnlNode);
SonI = (GNL_NODE)BListElt (Sons, 0);

if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar, SonI, ListCubes,
                               MaxCubes, Flattened))
    return (GNL_MEMORY_FULL);

if (!(*Flattened))
    return (GNL_OK);

*Flattened = 1;
if (BListSize (*ListCubes) == 0) /* Actually the 0 Constante */
    return (GNL_OK);

for (i=1; i<BListSize (Sons); i++)
{
    SonI = (GNL_NODE)BListElt (Sons, i);
    if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar, SonI,
                                    &ListCubesSon, MaxCubes, Flattened))
        return (GNL_MEMORY_FULL);

    if (!(*Flattened))
        return (GNL_OK);

    if (BListSize (ListCubesSon) == 0)
    {
        BListDelete (ListCubes, GnlCubeFree);
        *ListCubes = ListCubesSon;
        return (GNL_OK);
    }

    NbCubes1 = BListSize (*ListCubes);
    NbCubes2 = BListSize (ListCubesSon);

    if (GnlCubesListMultiplyFlattenable (NbCells, *ListCubes,
                                         ListCubesSon, MaxCubes, &ListRes))
        return (GNL_MEMORY_FULL);

    if (BListSize (ListRes) > MaxCubes)
    {
        BListDelete (&ListRes, GnlCubeFree);
        *Flattened = 0;
        return (GNL_OK);
    }

/* If the multiplication of the lists of cubes produces a list */
/* too large (e.g. of size 2 times greater than the sum of the */
/* two original lists) then we invoke the two-level minimization*/
if (BListSize (ListRes) > 2 * (NbCubes1 + NbCubes2))
{
    if (LsMinListCubes (ListRes, NULL, NbVar, &MinimizedListRes))
        return (GNL_MEMORY_FULL);
```

gnlcube.c

```
    BListDelete (&ListRes, GnlCubeFree);
    ListRes = MinimizedListRes;
}
BListDelete (ListCubes, GnlCubeFree);
BListDelete (&ListCubesSon, GnlCubeFree);
*ListCubes = ListRes;

if (BListSize (ListRes) == 0)      /* Actually the 0 Constante */
{
    *Flattened = 1;
    return (GNL_OK);
}

}

*Flattened = 1;
return (GNL_OK);
}

/*-----
/* GnlCubeNot
*/
-----*/
GNL_STATUS GnlCubeNot (NbCells, Cube, CubeNotList)
    int          NbCells;
    GNL_CUBE    Cube;
    BLIST       *CubeNotList;
{
    BLIST     ComplementList;
    GNL_CUBE  NewCube;
    unsigned   *VarSet;
    int         CellNb;
    unsigned   VarMask;

    if (BListCreate (&ComplementList))
        return (GNL_MEMORY_FULL);

    if (MintXNOR (GnlCubeHigh (Cube), GnlCubeLow (Cube), NbCells, &VarSet))
        return (GNL_MEMORY_FULL);

    while (MintLocateFirstOne (VarSet, NbCells, &CellNb, &VarMask))
    {
        if (GnlCubeFullAndCreate (NbCells, &NewCube))
            return (GNL_MEMORY_FULL);

        if (BListAddElt (ComplementList, NewCube))
            return (GNL_MEMORY_FULL);

        if (GnlCubeHigh(Cube) [CellNb] & VarMask)
            GnlCubeLow(NewCube) [CellNb] &= ~VarMask;
        else
            GnlCubeHigh(NewCube) [CellNb] |= VarMask;

        VarSet [CellNb] &= ~VarMask;
    }
}
```

gnlcube.c

```
}

free ((char *)VarSet);

*CubeNotList = ComplementList;

return (GNL_OK);
}

/*-----
/* GnlCubeComplementListCubes
*/
-----*/
GNL_STATUS GnlCubeComplementListCubes (Gnl, NbCells, NbVar, ListCubes,
                                         ListCubesNot)
{
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    BLIST        ListCubes;
    BLIST        *ListCubesNot;
{
    int          i;
    BLIST        OldCompList;
    BLIST        NewCompList;
    BLIST        CubeNotList ;
    GNL_CUBE    NewCube;
    GNL_CUBE    CubeI;

    if (BListCreateWithSize (1, &OldCompList))
        return (GNL_MEMORY_FULL);

    /* The first Cube is set to "----"
    if (GnlCubeFullAndCreate (NbCells, &NewCube))           */
        return (GNL_MEMORY_FULL);

    if (BListAddElt (OldCompList, (int)NewCube))
        return (GNL_MEMORY_FULL);

    /* if BListSize (ListCubes) is 0 it is Constant 0, the following loop */
    /* is not entered.                                                 */
    /*                                                               */

    for (i = 0; i < BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        if (GnlCubeNot (NbCells, CubeI, &CubeNotList))
            return (GNL_MEMORY_FULL);

        if (GnlCubesListMultiply (NbCells, OldCompList, CubeNotList,
                                  &NewCompList))
            return (GNL_MEMORY_FULL);

        BListDelete (&OldCompList, GnlCubeFree);
        BListDelete (&CubeNotList, GnlCubeFree) ;
        OldCompList = NewCompList;
    }
}
```

gnlcube.c

```
*ListCubesNot = OldCompList;

    return (GNL_OK);
}

/*-----
/* GnlCubeComplementListCubesFlattenable           */
/*-----*/
GNL_STATUS GnlCubeComplementListCubesFlattenable (Gnl, NbCells, NbVar,
                                                ListCubes, MaxCubes,
                                                ListCubesNot)

GNL          Gnl;
int          NbCells;
int          NbVar;
BLIST        ListCubes;
int          MaxCubes;
BLIST        *ListCubesNot;

{
    int          i;
    BLIST        OldCompList;
    BLIST        NewCompList;
    BLIST        CubeNotList ;
    GNL_CUBE    NewCube;
    GNL_CUBE    CubeI;

    if (BListCreateWithSize (1, &OldCompList))
        return (GNL_MEMORY_FULL);

    /* The first Cube is set to "----"                         */
    if (GnlCubeFullAndCreate (NbCells, &NewCube))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (OldCompList, (int)NewCube))
        return (GNL_MEMORY_FULL);

    /* if BListSize (ListCubes) is 0 it is Constant 0, the following loop */
    /* is not entered.                                                 */
    /*-----*/

    for (i = 0; i < BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        if (GnlCubeNot (NbCells, CubeI, &CubeNotList))
            return (GNL_MEMORY_FULL);

        if (GnlCubesListMultiply (NbCells, OldCompList, CubeNotList,
                                  &NewCompList))
            return (GNL_MEMORY_FULL);

        BListDelete (&OldCompList, GnlCubeFree);
        BListDelete (&CubeNotList, GnlCubeFree) ;
        OldCompList = NewCompList;

        /* we stop since we reach the max number of allowed cubes      */
        if (BListSize (OldCompList) > MaxCubes)
            break;
    }
}
```

gnlcube.c

```
}

*ListCubesNot = OldCompList;

return (GNL_OK);
}

/*-----
/* GnlCubesFromGnlNodeNOT
*/
/*-----*/
GNL_STATUS GnlCubesFromGnlNodeNOT (Gnl, NbCells, NbVar, GnlNode, ListCubes)
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST       *ListCubes;
{
    BLIST      Sons;
    GNL_NODE   Son;
    BLIST      ListCubesSon;
    GNL_CUBE   CubeOne;

Sons = GnlNodeSons (GnlNode);
Son = (GNL_NODE)BListElt (Sons, 0);
if (GnlCubesFromGnlNode (Gnl, NbCells, NbVar, Son, &ListCubesSon))
    return (GNL_MEMORY_FULL);

if (BListSize (ListCubesSon) == 0)
{
    if (BListCreate (ListCubes))
        return (GNL_MEMORY_FULL);

    if (GnlCubeFullAndCreate (NbCells, &CubeOne))
        return (GNL_MEMORY_FULL);

    if (BListAddElt ((*ListCubes), (int)CubeOne))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

if (GnlCubeComplementListCubes (Gnl, NbCells, NbVar, ListCubesSon,
                                ListCubes))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* GnlMakeFlattenableNodeNOT
*/
/*-----*/
GNL_STATUS GnlMakeFlattenableNodeNOT (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes, MaxCubes, Flattened)
    GNL          Gnl;
    int          NbCells;
```

gnlcube.c

```
int          NbVar;
GNL_NODE    GnlNode;
BLIST      *ListCubes;
int          MaxCubes;
int          *Flattened;
{
    BLIST    Sons;
    GNL_NODE Son;
    BLIST    ListCubesSon;
    GNL_CUBE CubeOne;

    Sons = GnlNodeSons (GnlNode);
    Son = (GNL_NODE)BListElt (Sons, 0);
    if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar, Son,
                                    &ListCubesSon, MaxCubes, Flattened))
        return (GNL_MEMORY_FULL);

    if (!(*Flattened))
        return (GNL_OK);

    if (BListSize (ListCubesSon) == 0)
    {
        *Flattened = 1;
        if (BListCreate (ListCubes))
            return (GNL_MEMORY_FULL);

        if (GnlCubeFullAndCreate (NbCells, &CubeOne))
            return (GNL_MEMORY_FULL);

        if (BListAddElt ((*ListCubes), (int)CubeOne))
            return (GNL_MEMORY_FULL);

        return (GNL_OK);
    }

    if (GnlCubeComplementListCubesFlattenable (Gnl, NbCells, NbVar,
                                                ListCubesSon, MaxCubes, ListCubes))
        return (GNL_MEMORY_FULL);

    if (BListSize (*ListCubes) > MaxCubes)
    {
        *Flattened = 0;
        return (GNL_OK);
    }

    *Flattened = 1;
    return (GNL_OK);
}

/*-----
/* GnlCubesFromGnlNodeCONSTANTE
/*-----*/
GNL_STATUS GnlCubesFromGnlNodeCONSTANTE (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes)
GNL          Gnl;
int          NbCells;
```

gnlcube.c

```
int NbVar;
GNL_NODE GnlNode;
BLIST *ListCubes;
{
    int Value;
    GNL_CUBE NewCube;

    Value = (int)GnlNodeSons (GnlNode);

    if (BListCreateWithSize (1, ListCubes))
        return (GNL_MEMORY_FULL);

    /* The constant value is 1 otherwise it is 0. */
    if (Value == 1)
    {
        if (GnlCubeFullAndCreate (NbCells, &NewCube))
            return (GNL_MEMORY_FULL);
        if (BListAddElt ((*ListCubes), (int)NewCube))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlCubesFromGnlNodeVARIABLE */
/*-----*/
/* WARNING: field 'GnlVarRank' of the variable must be correctly set */
/* according to the list inputs/outputs/locals of 'Gnl'. */
/*-----*/
GNL_STATUS GnlCubesFromGnlNodeVARIABLE (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes)
{
    GNL Gnl;
    int NbCells;
    int NbVar;
    GNL_NODE GnlNode;
    BLIST *ListCubes;
    GNL_VAR Var;
    int Rank;
    int RankInCell;
    GNL_CUBE NewCube;

    Var = (GNL_VAR)GnlNodeSons (GnlNode);

    /* case where Var is a VDD or a VSS. */
    if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
    {
        if (BListCreateWithSize (1, ListCubes))
            return (GNL_MEMORY_FULL);

        if (GnlVarIsVss (Var))
            return (GNL_OK);
```

gnlcube.c

```
if (GnlCubeFullAndCreate (NbCells, &NewCube))
    return (GNL_MEMORY_FULL);
if (BListAddElt ((*ListCubes), (int)NewCube))
    return (GNL_MEMORY_FULL);
return (GNL_OK);
}

Rank = GnlVarRank (Var); /* The Rank of the var in the list */
/* {Inputs+Outputs+Locals} */

if (GnlCubeFullAndCreate (NbCells, &NewCube))
    return (GNL_MEMORY_FULL);

locate (Rank, BITS_PER_INT, NbCells, RankInCell);

(GnlCubeHigh (NewCube)) [NbCells] |= (1 << (RankInCell -1));
(GnlCubeLow (NewCube)) [NbCells] |= (1 << (RankInCell -1));

if (BListCreateWithSize (1, ListCubes))
    return (GNL_MEMORY_FULL);

if (BListAddElt ((*ListCubes), (int)NewCube))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlCubesFromGnlNode
 */
GNL_STATUS GnlCubesFromGnlNode (Gnl, NbCells, NbVar, GnlNode, ListCubes)
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST       *ListCubes;
{
    switch (GnlNodeOp (GnlNode)) {
        case GNL_OR:
            return (GnlCubesFromGnlNodeOR (Gnl, NbCells, NbVar, GnlNode,
                ListCubes));

        case GNL_AND:
            return (GnlCubesFromGnlNodeAND (Gnl, NbCells, NbVar, GnlNode,
                ListCubes));

        case GNL_NOT:
            return (GnlCubesFromGnlNodeNOT (Gnl, NbCells, NbVar, GnlNode,
                ListCubes));

        case GNL_CONSTANTE:
            return (GnlCubesFromGnlNodeCONSTANTE (Gnl, NbCells, NbVar,
                GnlNode, ListCubes));

        case GNL_VARIABLE:
```

gnlcube.c

```
        return (GnlCubesFromGnlNodeVARIABLE (Gnl, NbCells, NbVar,
                                              GnlNode, ListCubes)) ;

    default:
        GnlError (12 /* unknow operator */);
        return (1);
    }

}

/*-----*/
/* GnlMakeFunctionFlattenable */
/*-----*/
GNL_STATUS GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes, MaxCubes, Flattened)
{
    GNL          Gnl;
    int          NbCells;
    int          NbVar;
    GNL_NODE    GnlNode;
    BLIST      *ListCubes;
    int          MaxCubes;
    int          *Flattened;
}

if (!GnlNode)
{
    *Flattened = 1;
    return (GNL_OK);
}

switch (GnlNodeOp (GnlNode)) {
    case GNL_OR:
        return (GnlMakeFlattenableNodeOR (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes, MaxCubes, Flattened));

    case GNL_AND:
        return (GnlMakeFlattenableNodeAND (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes, MaxCubes, Flattened));

    case GNL_NOT:
        return (GnlMakeFlattenableNodeNOT (Gnl, NbCells, NbVar, GnlNode,
                                         ListCubes, MaxCubes, Flattened));

    case GNL_CONSTANTE:
        *Flattened = 1;
        if (GnlCubesFromGnlNodeCONSTANTE (Gnl, NbCells, NbVar,
                                           GnlNode, ListCubes))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_VARIABLE:
        *Flattened = 1;
        if (GnlCubesFromGnlNodeVARIABLE (Gnl, NbCells, NbVar,
                                         GnlNode, ListCubes))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
}
```

gnlcube.c

```
default:
    GnlError (12 /* unknow operator */);
    return (1);
}

}

/*-----
/* UpdateVarRank
*/
-----*/
int UpdateVarRank (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR    VarI;

    for (i=0; i<BListSize (GnlInputs(Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlInputs(Gnl), i);
        SetGnlVarRank (VarI, i+1);
    }

    for (i=0; i<BListSize (GnlOutputs(Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlOutputs(Gnl), i);
        SetGnlVarRank (VarI, i+1+BListSize (GnlInputs(Gnl)));
    }

    for (i=0; i<BListSize (GnlLocals(Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlLocals(Gnl), i);
        SetGnlVarRank (VarI, i+1+BListSize (GnlInputs(Gnl))+
                        BListSize (GnlOutputs(Gnl)));
    }

    return (BListSize (GnlInputs(Gnl)) + BListSize (GnlOutputs(Gnl)) +
            BListSize (GnlLocals(Gnl)) + 1);
}

/*-----
/* GnlComputeCubesInGnl
*/
-----*/
GNL_STATUS GnlComputeCubesInGnl (Gnl)
    GNL      Gnl;
{
    int          NbVar;
    int          NbCells;
    int          i;
    GNL_FUNCTION FunctionI;
    BLIST       ListOnSetCubes;
    BLIST       ListDCSetCubes;
    GNL_VAR    VarI;
    int          NextRank;
```

gnlcube.c

```
/* number of variables in the design. */  
NbVar = BListSize (GnlInputs (Gnl)) + BListSize (GnlOutputs (Gnl)) +  
BListSize (GnlLocals (Gnl));  
  
/* Number of Ints to code the variables. */  
NbCells = NbOfCells (NbVar, BITS_PER_INT);  
  
/* We compute Rank of each Variable {inputs+outputs+locals} */  
NextRank = UpdateVarRank (Gnl);  
  
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)  
{  
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  
    FunctionI = GnlVarFunction (VarI);  
  
    /* Computing the list of cubes for the On set node. */  
    if (BListCreateWithSize (1, &ListOnSetCubes))  
        return (GNL_MEMORY_FULL);  
    if (GnlCubesFromGnlNode (Gnl, NbCells, NbVar,  
                            GnlFunctionOnSet (FunctionI),  
                            &ListOnSetCubes))  
        return (GNL_MEMORY_FULL);  
    SetGnlFunctionOnSetCubes (FunctionI, ListOnSetCubes);  
  
#ifdef TRACE  
    /* Printing the cubes ... */  
    GnlListCubesPrint (stderr, NbVar,  
                      GnlFunctionOnSetCubes (FunctionI));  
#endif  
  
    /* Computing the list of cubes for the DC set node. */  
    if (BListCreateWithSize (1, &ListDCSetCubes))  
        return (GNL_MEMORY_FULL);  
    if (GnlFunctionDCSet (FunctionI) &&  
        GnlCubesFromGnlNode (Gnl, NbCells, NbVar,  
                            GnlFunctionDCSet (FunctionI),  
                            &ListDCSetCubes))  
        return (GNL_MEMORY_FULL);  
    SetGnlFunctionDCSetCubes (FunctionI, ListDCSetCubes);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlCreateNewFunctionFromFunction */  
/*-----*/  
GNL_STATUS GnlCreateNewFunctionFromFunction (Gnl, NbVar, NbCells, Node)  
{  
    GNL          Gnl;  
    int          *NbVar;  
    int          *NbCells;  
    GNL_NODE    Node;  
    {  
        int          i;
```

gnlcube.c

```
GNL_NODE SonI;
BLIST Sons;
GNL_VAR NewVar;
GNL_NODE NewNode;
GNL_FUNCTION NewFunction;

if (!Node)
    return (GNL_OK);

switch (GnlNodeOp (Node)) {
    case GNL_CONSTANTE:
    case GNL_VARIABLE:
        return (GNL_OK);

    case GNL_NOT:
        (*NbVar)++;
        *NbCells = NbOfCells (*NbVar, BITS_PER_INT);
        Sons = GnlNodeSons (Node);
        SonI = (GNL_NODE)BListElt (Sons, 0);
        if (GnlCreateNewFunctionFromFunction (Gnl, NbVar, NbCells,
SonI))
            return (GNL_MEMORY_FULL);

        if (GnlCreateUniqueVar (Gnl, "D", &NewVar))
            return (GNL_MEMORY_FULL);
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (NewNode, (BLIST)NewVar);
        BListElt (Sons, 0) = (int)NewNode;
        if (GnlFunctionCreate (Gnl, NewVar, SonI, &NewFunction))
            return (GNL_MEMORY_FULL);
        SetGnlVarFunction (NewVar, NewFunction);

        if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
        if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlVarRank (NewVar, G_Rank++);

        return (GNL_OK);

    case GNL_OR:
    case GNL_AND:
        Sons = GnlNodeSons (Node);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlCreateNewFunctionFromFunction (Gnl, NbVar, NbCells,
SonI))
                return (GNL_MEMORY_FULL);
        }

        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
```

gnlcube.c

```
if (GnlNodeOp (SonI) == GNL_VARIABLE)
    continue;
if (GnlNodeOp (SonI) == GNL_CONSTANTE)
    continue;

(*NbVar)++;
*NbCells = NbOfCells (*NbVar, BITS_PER_INT);
if (GnlCreateUniqueVar (Gnl, "D", &NewVar))
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NewNode, (BLIST) NewVar);
BListElt (Sons, i) = (int) NewNode;
if (GnlFunctionCreate (Gnl, NewVar, SonI, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (NewVar, NewFunction);
if (BListAddElt (GnlLocals (Gnl), (int) NewVar))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
if (BListAddElt (GnlFunctions (Gnl), (int) NewVar))
    return (GNL_MEMORY_FULL);
SetGnlVarRank (NewVar, G_Rank++);
}

return (GNL_OK);
}

}

/*-----*/
/* GnlMakeGnlFlattenable */
/*-----*/
GNL_STATUS GnlMakeGnlFlattenable (Gnl, MaxCubes, Flattened)
    GNL          Gnl;
    int          MaxCubes;
    int          *Flattened;
{
    int          NbVar;
    int          NbCells;
    int          i;
    GNL_FUNCTION FunctionI;
    BLIST        ListOnSetCubes;
    BLIST        ListDCSetCubes;
    GNL_VAR      VarI;
    int          NbFunctions;

    /* number of variables in the design. */
    NbVar = BListSize (GnlInputs (Gnl)) + BListSize (GnlOutputs (Gnl)) +
            BListSize (GnlLocals (Gnl));

    /* Number of Ints to code the variables. */
    NbCells = NbOfCells (NbVar, BITS_PER_INT);

    /* We compute Rank of each Variable {inputs+outputs+locals} */
    G_Rank = UpdateVarRank (Gnl);
    */

}
```

gnlcube.c

```
NbFunctions = BListSize (GnlFunctions (Gnl));
for (i=0; i<NbFunctions; i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    /* Computing the list of cubes for the On set node. */
    if (BListCreateWithSize (1, &ListOnSetCubes))
        return (GNL_MEMORY_FULL);
    if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar,
                                    GnlFunctionOnSet (FunctionI),
                                    &ListOnSetCubes, MaxCubes, Flattened))
        return (GNL_MEMORY_FULL);
    if (!(*Flattened))
    {
        if (GnlCreateNewFunctionFromFunction (Gnl, &NbVar, &NbCells,
                                            GnlFunctionOnSet (FunctionI)))
            return (GNL_MEMORY_FULL);
    }
    SetGnlFunctionOnSetCubes (FunctionI, ListOnSetCubes);

    /* Computing the list of cubes for the DC set node. */
    if (BListCreateWithSize (1, &ListDCSetCubes))
        return (GNL_MEMORY_FULL);
    if (GnlMakeFunctionFlattenable (Gnl, NbCells, NbVar,
                                    GnlFunctionDCSet (FunctionI),
                                    &ListDCSetCubes, MaxCubes, Flattened))
        return (GNL_MEMORY_FULL);
    if (!(*Flattened))
    {
        if (GnlCreateNewFunctionFromFunction (Gnl, &NbVar, &NbCells,
                                            GnlFunctionDCSet (FunctionI)))
            return (GNL_MEMORY_FULL);
    }
    SetGnlFunctionDCSetCubes (FunctionI, ListDCSetCubes);
}

if (NbFunctions != BListSize (GnlFunctions (Gnl)))
{
    for (i=0; i<NbFunctions; i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        ListOnSetCubes = GnlFunctionOnSetCubes (FunctionI);
        BListDelete (&ListOnSetCubes, GnlCubeFree);
        SetGnlFunctionOnSetCubes (FunctionI, NULL);
        ListDCSetCubes = GnlFunctionDCSetCubes (FunctionI);
        BListDelete (&ListDCSetCubes, GnlCubeFree);
        SetGnlFunctionDCSetCubes (FunctionI, NULL);
    }
    *Flattened = 0;
    return (GNL_OK);
}

*Flattened = 1;
return (GNL_OK);
}
```

gnlcube.c

```
/*-----*/
```

gnlcube.h

```
/*----- */
/*                                         */
/*      File:          gnlcube.h           */
/*      Version:        1.1               */
/*      Modifications: -                */
/*      Documentation: -                */
/*                                         */
/*----- */
#ifndef GNLCUBE_H
#define GNLCUBE_H

/* ----- */
/* Macros */
/* ----- */

#define DIRECT_INSTANTIATE ~0
#define INVERSE_INSTANTIATE ~DIRECT_INSTANTIATE

#define GnlCubeNonVars(High, Low)    ((High)^(Low))
#define GnlCubeVars(High, Low)       (~GnlCubeNonVars(High, Low))

#define GnlCubeSignificantVars(High, Low, VarSet) \
    (GnlCubeVars(High, Low) & VarSet)

#define GnlCubeVarClash(High1, Low1, High2, Low2, DirectInverse) \
    ~(GnlCubeNonVars(High1, Low1) | \
      GnlCubeNonVars(High2, Low2) | \
      (GnlCubeNonVars(High1, High2)^(DirectInverse)))

#define GnlCubeONVars(High, Low)     ((High)&(Low))
#define GnlCubeOFFVars(High, Low)   (~((High) | (Low)))

#define GnlCubeSignificantONVars(High, Low, VarSet) \
    (GnlCubeONVars(High, Low) & VarSet)
#define GnlCubeSignificantOFFVars(High, Low, VarSet) \
    (GnlCubeOFFVars(High, Low) & VarSet)

#define IntDiff(x,y) ((x) > (y) ? (x) - (y) : (y) - (x))

/* ----- EOF ----- */
#endif
```

gnlec.c

```
/*----- */
/*          File:      gnlec.c           */
/*          Version:    1.1             */
/*          Modifications: -          */
/*          Documentation: -         */
/*          Class: none            */
/*          Inheritance:          */
/*----- */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "blist.h"
#include "blist.e"
#include "gnl.h"
#include "bbdd.h"
#include "gnlec.h"
#include "skiplist.h"
#include "gnloption.h"

/* ----- */
/* DEFINE                                         */
/* ----- */

#define STRUCT_MODE          0
#define REGULAR_MODE         1
#define BREAK_MODE           2
#define LOCAL_MODE           3
#define RANDOM_MODE          4

#define RANDOM_IN_VECTORS_NUM 32
#define MAX_CHECKED_LIST_SIZE 2

#define CHECK_MAX_CONFLICTS 1000000

/* #define ALL_CUTPOINTS_TOGETHER      Bdds for all CP built together */
/* #define COMMON_CUTPOINTS_HEUR      Uncommon CP composed first */
/* #define LEAST_BDD_HEUR           least Bdd composed first */
#define COMPOSE_UNTIL_FIRST_FAILURE /* Compose CP only until first fail */
/* #define PRINT_COMPOSE_STAT        Print Compose statistics */
/* #define PRINT_CUTPOINTS_STAT      Print CP statistics */
#define STRUCTURAL_CHECK         /* Structural Checking phase */
#define FND                      /* False Negative Detection phase */
/* #define FND_2                   Final False Negative Detection */
#define RANDOM_SIMULATION        /* Random Simulation phase */
/* #define XOR_ANALYSIS            XOR Analysis phase */
/* #define PRINT_BDD_BUILD          */
/* #define BFS_BDD_PHASE           BFS BDD building */
/* #define SPLITTING_NODES          Splitting multiple input nodes */
/* #define PRINT_SIMULATION_BUCKETS Print possibly equiv nodes lists */
/* #define IMPLICATION_CHECK       Checking implications of nodes */

/* ----- */
/* EXTERN                                         */
/* ----- */
```

gnlec.c

```
extern BDD_WS    GLOB_BDD_WS;      /* Current global BDD work space      */
extern GNL_ENV   G_GnlEnv;

/* -----
/* STATIC VARIABLES
/* ----- */

static int  G_EcUniqueId = 0;
static int  G_NbCutVar = 0;
static int  G_GnlMaxBddNode;
static int  G_FirstBrkVarIndex;
static BLIST   G_ListAllCutPoints;
static int  G_BridgeNodes = 0;
static int  G_InvBridgeNodes = 0;
static int  G_Tag = 0;

#ifndef COMMON_CUTPOINTS_HEUR
static BLIST   G_ListOut1CutPoints;
static BLIST   G_ListOut2CutPoints;
static BLIST   G_ListCommonCutPoints;
#endif

#ifndef RANDOM_SIMULATION
static BLIST   G_InVectors;
static SKIPLIST G_PossiblyEquiv = NULL;
static int     G_NbEquiv;
static int     G_SimulationNum = 0;
static int     G_DiffFound;
static int     G_EquivFound;
#endif

#ifndef XOR_ANALYSIS
static int     G_OnePath;
static int     G_NbOnePath;
static int     G_FalsePath = 1000000;
static BLIST   G_TransitiveXorSupport;
static BLIST   G_ListConflictBdd;
static int     G_CheckConflict;
#endif

#ifndef BFS_BDD_PHASE
static int     G_WsNum;
#endif

static BLIST   G_Outputs1;
static BLIST   G_Outputs2;
static BLIST   G_OutputNodes1;
static BLIST   G_OutputNodes2;

static BLIST   G_ListConstraints = NULL;
static BLIST   G_ListBadPatterns = NULL;
static int     G_BadPatternsFound;

#ifndef SPLITTING_NODES
/*-----
/* ComputeMaxInput
/*-----*/
void ComputeMaxInput (Node)
```

gnlec.c

```
    GNL_NODE      Node;
{
    int          i;
    int          MaxId = -1;
    GNL_NODE     SonI;

    if (GnlNodeHook (Node))
        return;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        SetGnlNodeHook (Node, (void *)-1);
        return;
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        SetGnlNodeHook (Node, (void *)GnlVarId ((GNL_VAR)GnlNodeSons (Node)));
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

        ComputeMaxInput (SonI);

        if (MaxId < (int)GnlNodeHook (SonI))
            MaxId = (int)GnlNodeHook (SonI);
    }

    SetGnlNodeHook (Node, (void *)MaxId);
}

/*-----*/
/* HookCmp                                         */
/*-----*/
int HookCmp (Node1, Node2)
    GNL_NODE      *Node1;
    GNL_NODE      *Node2;
{
    return ((int)GnlNodeHook (*Node1) - (int)GnlNodeHook (*Node2));
}

/*-----*/
/* ResetHookFields                                */
/*-----*/
void ResetHookFields (Node)
    GNL_NODE      Node;
{
    int          i;

    if (!GnlNodeHook (Node))
        return;

    SetGnlNodeHook (Node, NULL);
```

gnlec.c

```
if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
    (GnlNodeOp (Node) == GNL_VARIABLE))
    return;

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    ResetHookFields ((GNL_NODE)BlistElt (GnlNodeSons (Node), i));
}

/*-----
/* SplitNode
/*-----*/
GNL_STATUS SplitNode (RefGnl, Node)
    GNL          RefGnl;
    GNL_NODE     Node;
{
    int          i;
    int          NbSons;
    int          MaxHook;
    GNL_NODE    CurrSon;
    GNL_NODE    NextSon;
    GNL_NODE    NewSon;
    BLIST       NewSonsList;

    if (GnlNodeTag (Node) == G_Tag)
        return (GNL_OK);

    SetGnlNodeTag (Node, G_Tag);

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return (GNL_OK);

    NbSons = BListSize (GnlNodeSons (Node));

    for (i=0; i<NbSons; i++)
        if (SplitNode (RefGnl, (GNL_NODE)BlistElt (GnlNodeSons (Node), i)))
            return (GNL_MEMORY_FULL);

    if (NbSons <= 2)
        return (GNL_OK);

    qsort (BlistAdress (GnlNodeSons (Node)), NbSons, sizeof (GNL_NODE),
    HookCmp);

    i = 1;
    CurrSon = (GNL_NODE)BlistElt (GnlNodeSons (Node), 0);

    while (i < NbSons - 1)
    {
        NextSon = (GNL_NODE)BlistElt (GnlNodeSons (Node), i++);

        if (GnlCreateNode (RefGnl, GnlNodeOp (Node), &NewSon))
            return (GNL_MEMORY_FULL);

        if (BlistCreateWithSize (2, &GnlNodeSons (NewSon)))
            return (NULL);
```

```

BListAddElt (GnlNodeSons (NewSon), (int)CurrSon);
BListAddElt (GnlNodeSons (NewSon), (int)NextSon);

MaxHook = ((int)GnlNodeHook (CurrSon) < (int)GnlNodeHook (NextSon) ?
            (int)GnlNodeHook (NextSon) : (int)GnlNodeHook (CurrSon));

SetGnlNodeHook (NewSon, (void *)MaxHook);

CurrSon = NewSon;
}

if (BListCreateWithSize (2, &NewSonsList))
    return (GNL_MEMORY_FULL);

BListAddElt (NewSonsList, (int)CurrSon);
BListAddElt (NewSonsList, BListElt (GnlNodeSons (Node), NbSons - 1));

BListQuickDelete (&GnlNodeSons (Node));
SetGnlNodeSons (Node, NewSonsList);

return (GNL_OK);
}

/*-----
/* SplitNodesInGnl
/*-----*/
GNL_STATUS SplitNodesInGnl (RefGnl, ListOfOutNodes)
    GNL          RefGnl;
    BLIST        ListOfOutNodes;
{
    int          i;

    for (i=0; i<BListSize (ListOfOutNodes); i++)
        if (SplitNode (RefGnl, (GNL_NODE)BListElt (ListOfOutNodes, i)))
            return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
#endif

/*-----
/* BListAddEltWCheck
/*-----*/
static int BListAddEltWCheck (L, Elt)
    BLIST *L;
    int   Elt;
{
    if (*L == NULL)
        if (BListCreate (L))
            return (1);
        return (BListAddElt (*L, Elt));
}

/*-----
/* GnlVarOnSetNode
/*-----*/
/* PREC: before invoking this function the Dads field of all primary */

```

gnlec.c

```
/*      inputs has to be assigned (in our case by RemoveLocals proc.) */
/*-----*/
GNL_NODE GnlVarOnSetNode (Var)
    GNL_VAR      Var;
{
    GNL_FUNCTION  Func;

    if (GnlVarDir (Var) == GNL_VAR_INPUT)
        return ((GNL_NODE) GnlVarDads (Var));

    if ((Func = GnlVarFunction (Var)) == NULL)
        return (NULL);
    return (GnlFunctionOnSet (Func));
}

/* ----- */
/* AllocateHookPtrs                                */
/*-----*/
GNL_STATUS AllocateHookPtrs (Node)
    GNL_NODE      Node;
{
    GNL_CONSTRAINT NewConstraint;
    BLIST         Signatures;

    if ((NewConstraint = (GNL_CONSTRAINT) calloc (1, sizeof
(GNL_CONSTRAINT_REC))) == NULL)
        return (GNL_MEMORY_FULL);
    SetGnlConstraintNode (NewConstraint, Node);
    SetGnlConstraintValue (NewConstraint, 0);
    SetGnlNodeNegConstraint (Node, NewConstraint);

    if ((NewConstraint = (GNL_CONSTRAINT) calloc (1, sizeof
(GNL_CONSTRAINT_REC))) == NULL)
        return (GNL_MEMORY_FULL);
    SetGnlConstraintNode (NewConstraint, Node);
    SetGnlConstraintValue (NewConstraint, 1);
    SetGnlNodePosConstraint (Node, NewConstraint);

#ifndef RANDOM_SIMULATION
    if (BListCreate (&Signatures))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSignatures (Node, Signatures);
#endif

    return (GNL_OK);
}

/*-----*/
/* CreateNodeHookField                            */
/*-----*/
GNL_STATUS CreateNodeHookField (Node)
    GNL_NODE      Node;
{
    int           i;
    void          *Ptr;

    if (!Node || GnlNodeHook (Node))
```

```

        return (GNL_OK);

    if ((GnlNodeOp (Node) != GNL_CONSTANTE) &&
        (GnlNodeOp (Node) != GNL_VARIABLE))
        for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
            if (CreateNodeHookField ((GNL_NODE)BListElt (GnlNodeSons (Node), i)))
                return (GNL_MEMORY_FULL);

    if ((Ptr = calloc (1, sizeof (ADD_GNL_NODE_REC))) == NULL)
        return (GNL_MEMORY_FULL);
    SetGnlNodeHook (Node, Ptr);

    if (GnlNodeOp (Node) != GNL_CONSTANTE)
        if (AllocateHookPtrs (Node))
            return (GNL_MEMORY_FULL);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
        SetGnlNodeVar (Node, (GNL_VAR)GnlNodeSons (Node));

    return (GNL_OK);
}

/*-----
/* CreateNodesHookFields
/*-----*/
GNL_STATUS CreateNodesHookFields (Gnl1, Gnl2)
{
    GNL          Gnl1;
    GNL          Gnl2;
{
    int          i;

    for (i=0; i<BListSize (G_OutputNodes1); i++)
        if (CreateNodeHookField ((GNL_NODE)BListElt (G_OutputNodes1, i)))
            return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (G_OutputNodes2); i++)
        if (CreateNodeHookField ((GNL_NODE)BListElt (G_OutputNodes2, i)))
            return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlLocals (Gnl1)); i++)
        if (CreateNodeHookField (GnlVarOnSetNode((GNL_VAR)BListElt (GnlLocals
(Gnl1), i))))
            return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlLocals (Gnl2)); i++)
        if (CreateNodeHookField (GnlVarOnSetNode((GNL_VAR)BListElt (GnlLocals
(Gnl2), i))))
            return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* CreateVarsHookFields
/*-----*/
GNL_STATUS CreateVarsHookFields (Gnl)
{
    GNL          Gnl;

```

```

{
    int          i;
    GNL_VAR     VarI;
    GNL_NODE    NodeI;
    void        *Ptr;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        if ((Ptr = calloc (1, sizeof (ADD_GNL_VAR_REC))) == NULL)
            return (GNL_MEMORY_FULL);
        SetGnlVarHook (VarI, Ptr);
        NodeI = GnlFunctionOnSet ((GNL_FUNCTION)GnlVarFunction (VarI));
        if (GnlNodeHook (NodeI) &&
            (GnlNodeOp (NodeI) != GNL_VARIABLE) &&
            (GnlNodeOp (NodeI) != GNL_CONSTANTE))
            SetGnlNodeVar (NodeI, VarI);
    }

    for (i=0; i<BListSize (GnlRelevantInputs (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlRelevantInputs (Gnl), i);
        if (GnlVarHook (VarI))
            continue;
        if ((Ptr = calloc (1, sizeof (ADD_GNL_VAR_REC))) == NULL)
            return (GNL_MEMORY_FULL);
        SetGnlVarHook (VarI, Ptr);
    }

    for (i=0; i<2; i++)
        if (VarI = GnlConstant (Gnl, i))
        {
            if ((Ptr = calloc (1, sizeof (ADD_GNL_VAR_REC))) == NULL)
                return (GNL_MEMORY_FULL);
            SetGnlVarHook (VarI, Ptr);
            NodeI = (GNL_NODE)GnlVarDads (VarI);
            if (GnlNodeHook (NodeI))
                SetGnlNodeVar (NodeI, VarI);
        }

    return (GNL_OK);
}

/*-----*/
/* FreeNodesHook */
/*-----*/
void FreeNodesHook (Node)
    GNL_NODE    Node;
{
    free (GnlNodeNegConstraint (Node));
    SetGnlNodeNegConstraint (Node, NULL);
    free (GnlNodePosConstraint (Node));
    SetGnlNodePosConstraint (Node, NULL);
#endif RANDOM_SIMULATION
    BListDelete (&GnlNodeSignatures (Node), BListQuickDelete);
#endif
    free (GnlNodeHook (Node));
}

```

gnlec.c

```
    SetGnlNodeHook (Node, NULL);
}

/*-----*/
/* FreeNodeHookField                                */
/*-----*/
void FreeNodeHookField (Node)
    GNL_NODE      Node;
{
    int          i;

    if (!Node || !GnlNodeHook (Node))
        return;

    if (GnlNodeVar (Node))
    {
        free (GnlVarHook (GnlNodeVar (Node)));
        SetGnlVarHook (GnlNodeVar (Node), NULL);
    }

    FreeNodesHook (Node);

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) || (GnlNodeOp (Node) ==
GNL_VARIABLE))
        return;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        FreeNodeHookField ((GNL_NODE)BListElt (GnlNodeSons (Node), i));
}

/*-----*/
/* FreeHookFields                                */
/*-----*/
void FreeHookFields (Gnl1, Gnl2)
    GNL          Gnl1;
    GNL          Gnl2;
{
    int          i;

    for (i=0; i<BListSize (G_Outputs1); i++)
        FreeNodeHookField (GnlVarOnSetNode ((GNL_VAR)BListElt (G_Outputs1, i)));

    for (i=0; i<BListSize (G_Outputs2); i++)
        FreeNodeHookField (GnlVarOnSetNode ((GNL_VAR)BListElt (G_Outputs2, i)));

    for (i=0; i<BListSize (GnlLocals (Gnl1)); i++)
        FreeNodeHookField (GnlVarOnSetNode ((GNL_VAR)BListElt (GnlLocals (Gnl1),
i)));

    for (i=0; i<BListSize (GnlLocals (Gnl2)); i++)
        FreeNodeHookField (GnlVarOnSetNode ((GNL_VAR)BListElt (GnlLocals (Gnl2),
i)));
}

/*-----*/
/* MakeNewTag                                */
/*-----*/
```

gnlec.c

```
/* We update the tags of both netlists Gnl1 and Gnl2 with a new value */
/*-----*/
void MakeNewTag (Gnl1, Gnl2)
    GNL          Gnl1;
    GNL          Gnl2;
{
    int          MaxTag;

    MaxTag = (GnlTag (Gnl1) < GnlTag (Gnl2) ?
              GnlTag (Gnl2) : GnlTag (Gnl1));
    if (MaxTag < G_Tag)
        MaxTag = G_Tag;
    SetGnlTag (Gnl1, MaxTag+1);
    SetGnlTag (Gnl2, MaxTag+1);
    G_Tag = MaxTag+1;
}

/*-----*/
/* MakeNewTagBdd                               */
/*-----*/
/* We update the bdd tags of netlists Gnl1 and Gnl2 with a new value */
/*-----*/
void MakeNewTagBdd (Gnl1, Gnl2)
    GNL          Gnl1;
    GNL          Gnl2;
{
    int          MaxTag;

    MaxTag = (GnlTagBdd (Gnl1) < GnlTagBdd (Gnl2) ?
              GnlTagBdd (Gnl2) : GnlTagBdd (Gnl1));
    SetGnlTagBdd (Gnl1, MaxTag+1);
    SetGnlTagBdd (Gnl2, MaxTag+1);
}

/* ----- */
/* GnlCreateBddRefInfo                         */
/* ----- */
GNL_STATUS GnlCreateBddRefInfo (BddRefInfo)
    BDD_REF_INFO *BddRefInfo;
{

    if ((*BddRefInfo = (BDD_REF_INFO)malloc (1,
                                              sizeof(BDD_REF_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/* ----- */
/* GnlModifyEcNodeInfo                         */
/* ----- */
void GnlModifyEcNodeInfo (Node)
    GNL_NODE     Node;
{
    int          i;
    BDD          Bdd;
    BDD_PTR     BddPtr;
```

```

BDD_VAR          BddVar;

if (GnlNodeTag (Node) == G_Tag)
    return;
SetGnlNodeTag (Node, G_Tag);

if ((GnlNodeOp (Node) != GNL_CONSTANTE) &&
    (GnlNodeOp (Node) != GNL_VARIABLE))
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        GnlModifyEcNodeInfo ((GNL_NODE)BListElt (GnlNodeSons (Node), i));

if (Bdd = GnlNodeBdd (Node))
{
    BddPtr = GetBddPtr (Bdd);
    free ((BDD_REF_INFO)GetBddPtrHook (BddPtr));
    SetBddPtrHook (BddPtr, (Uint32)NULL);
}

if ((BddVar = GnlNodeBreakBddVar (Node)) &&
    (Bdd = BddVarBdd (BddVar)))
{
    BddPtr = GetBddPtr (Bdd);
    free ((BDD_REF_INFO)GetBddPtrHook (BddPtr));
    SetBddPtrHook (BddPtr, (Uint32)NULL);
}

if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return;

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    SetGnlVarBdd (GnlNodeVar (Node), BDD_NULL);
    SetGnlNodeBdd (Node, BDD_NULL);
    return;
}

/* Modifying also the Node information. */
if (GnlNodeBreakBddVar (Node))
    SetGnlNodeBreakBddVar (Node, (BDD_VAR)-1);
SetGnlNodeBdd (Node, BDD_NULL);
}

/*
 * -----
 * GnlFreeWorkSpaceFromNodes
 * -----
 */
void GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2)
    BDD_WS           Ws;
    GNL_NODE         Node1;
    GNL_NODE         Node2;
{
    G_Tag++;
    GnlModifyEcNodeInfo (Node1);
    GnlModifyEcNodeInfo (Node2);
    FreeBddWorkSpace (Ws);
}

/*
 * -----
 */

```

gnlec.c

```
/* GnlCutPointFree */  
/* ----- */  
void GnlCutPointFree (Cp)  
    GNL_EC_CUT_POINT      *Cp;  
{  
    free (*Cp);  
    *Cp = NULL;  
}  
  
/* ----- */  
/* GnlCreateEcCutPoint */  
/* ----- */  
GNL_STATUS GnlCreateEcCutPoint (EcCutPoint)  
    GNL_EC_CUT_POINT      *EcCutPoint;  
{  
  
    if ((*EcCutPoint = (GNL_EC_CUT_POINT) calloc (1,  
        sizeof(GNL_EC_CUT_POINT_REC))) == NULL)  
        return (GNL_MEMORY_FULL);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* VarInGnl */  
/*-----*/  
/*-----*/  
int VarInGnl (Var, Gnl)  
    GNL_VAR   Var;  
    GNL       Gnl;  
{  
    int      i;  
  
    if ((GnlConstant (Gnl, 0) == Var) || (GnlConstant (Gnl, 1) == Var))  
        return (1);  
    for (i=0; i<GnlNbLocal (Gnl); i++)  
        if ((GNL_VAR)BListElt (GnlLocals (Gnl), i) == Var)  
            return (1);  
    for (i=0; i<GnlNbIn (Gnl); i++)  
        if ((GNL_VAR)BListElt (GnlInputs (Gnl), i) == Var)  
            return (1);  
    for (i=0; i<GnlNbOut (Gnl); i++)  
        if ((GNL_VAR)BListElt (GnlOutputs (Gnl), i) == Var)  
            return (1);  
    return (0);  
}  
*/  
  
/*-----*/  
/* GetBuffInVar */  
/*-----*/  
GNL_VAR GetBuffInVar (Gnl, OutVar)  
    GNL           Gnl;  
    GNL_VAR       OutVar;  
{  
    int          i;  
    GNL_USER_COMPONENT CompI;
```

```

GNL_ASSOC          Port;

for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
{
    CompI = (GNL_USER_COMPONENT)BListElt (GnlComponents (Gnl), i);
    if ((GnlUserComponentType (CompI) == GNL_USER_COMPO) &&
        !strcmp (GnlUserComponentName (CompI), "buff") &&
        !strcmp (GnlUserComponentInstName (CompI), GnlVarName (OutVar)))
    {
        Port = (GNL_ASSOC)BListElt (GnlUserComponentInterface (CompI), 1);
        if (GnlVarIsVar (GnlAssocActualPort (Port)))
            return (GnlAssocActualPort (Port));
        else
            return (NULL);
    }
}

return (NULL);
}

/*-----
/* ComputeGnlValueOnNodeOR                               */
/*-----*/
void ComputeGnlValueOnNodeOR (Node)
    GNL_NODE Node;
{
    int           i;
    GNL_NODE NodeI;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 1)
        {
            SetGnlnodeValue (Node, 1);
            return;
        }
    }

    SetGnlnodeValue (Node, 0);
}

/*-----
/* ComputeGnlValueOnNodeNOR                            */
/*-----*/
void ComputeGnlValueOnNodeNOR (Node)
    GNL_NODE Node;
{
    int           i;
    GNL_NODE NodeI;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 1)
        {
            SetGnlnodeValue (Node, 0);
        }
    }
}

```

gnlec.c

```
        return;
    }
}

SetGnlnodeValue (Node, 1);
}

/*-----
/* ComputeGnlValueOnNodeAND
/*-----*/
void ComputeGnlValueOnNodeAND (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE NodeI;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 0)
        {
            SetGnlnodeValue (Node, 0);
            return;
        }
    }

    SetGnlnodeValue (Node, 1);
}

/*-----
/* ComputeGnlValueOnNodeNAND
/*-----*/
void ComputeGnlValueOnNodeNAND (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE NodeI;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 0)
        {
            SetGnlnodeValue (Node, 1);
            return;
        }
    }

    SetGnlnodeValue (Node, 0);
}

/*-----
/* ComputeGnlValueOnNodeNOT
/*-----*/
void ComputeGnlValueOnNodeNOT (Node)
    GNL_NODE Node;
{
```

gnlec.c

```
GNL_NODE NodeIn;

NodeIn = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);
SetGnlnodeValue (Node, 1-GnlnodeValue (NodeIn));
}

/*-----
/* ComputeGnlValueOnNodeXOR                               */
/*-----*/
void ComputeGnlValueOnNodeXOR (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE NodeI;
    int             NbOne = 0;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 1)
            NbOne++;
    }

    SetGnlnodeValue (Node, NbOne%2);
}

/*-----
/* ComputeGnlValueOnNodeXNOR                            */
/*-----*/
void ComputeGnlValueOnNodeXNOR (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE NodeI;
    int             NbOne = 0;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlnodeValue (NodeI) == 1)
            NbOne++;
    }

    SetGnlnodeValue (Node, 1-NbOne%2);
}

/*-----
/* ComputeGnlValueOnNode                               */
/*-----*/
void ComputeGnlValueOnNode (Node)
    GNL_NODE Node;
{
    switch (GnlNodeOp (Node))
    {
        case GNL_OR:
            ComputeGnlValueOnNodeOR (Node);
            return;
    }
}
```

```

    case GNL_NOR:
        ComputeGnlValueOnNodeNOR (Node);
        return;

    case GNL_AND:
        ComputeGnlValueOnNodeAND (Node);
        return;

    case GNL_NAND:
        ComputeGnlValueOnNodeNAND (Node);
        return;

    case GNL_NOT:
        ComputeGnlValueOnNodeNOT (Node);
        return;

    case GNL_XOR:
        ComputeGnlValueOnNodeXOR (Node);
        return;

    case GNL_XNOR:
        ComputeGnlValueOnNodeXNOR (Node);
        return;
    }

}

/*-----
/* SimulateGnlValue
/*-----*/
GNL_STATUS SimulateGnlValue (Gnl, Node, Index)
{
    GNL          Gnl;
    GNL_NODE     Node;
    int          Index; /* In the case of random simulation - number */
                        /* of input vector, otherwise -1 */
    int          i;
    GNL_NODE     NodeI;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return (GNL_OK);

    SetGnlNodeTag (Node, GnlTag (Gnl));

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        SetGnlNodeValue (Node, (GnlNodeSons (Node) ? 1 : 0));
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        if (GnlNodeValue (Node) == -1)
            SetGnlNodeValue (Node, 1);
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)

```

```

{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (SimulateGnlValue (Gnl, NodeI, Index))
        return (GNL_MEMORY_FULL);
}

ComputeGnlValueOnNode (Node);

#ifndef RANDOM_SIMULATION
if (Index >= 0)
{
    BLIST Signatures = GnlNodeSignatures (Node);
    BLIST CurrSignature;

    if (!Index)
    {
        if (BListCreateWithSize (BListSize (G_InVectors), &CurrSignature))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (Signatures, (int)CurrSignature))
            return (GNL_MEMORY_FULL);
    }
    else
        CurrSignature = (BLIST)BListElt (Signatures, BListSize (Signatures) -
1);

        BListAddElt (CurrSignature, GnlNodeValue (Node));
    }
#endif

    return (GNL_OK);
}

/*-----
/* ResetInputs
/*-----*/
void ResetInputs (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_NODE     InNodeI;

    for (i=0; i<BListSize (GnlRelevantInputs (Gnl)); i++)
    {
        if (InNodeI = (GNL_NODE)GnlVarDads ((GNL_VAR)BListElt
(GnlRelevantInputs (Gnl), i)))
            SetGnlnodeValue (InNodeI, -1);
    }
}

/*-----
/* SetInputs
/*-----*/
void SetInputs (BadPattern)
    BLIST          BadPattern;
{
    int           i;
    GNL_CONSTRAINT ConstraintI;
}

```

```

for (i=0; i<BListSize (BadPattern); i++)
{
    if (ConstraintI = (GNL_CONSTRAINT)BListElt (BadPattern, i))
        SetGnlNodeValue (GnlConstraintNode (ConstraintI),
                          GnlConstraintValue (ConstraintI));
}
}

/*-----*/
/* CreateInputNode */
/*-----*/
GNL_NODE CreateInputNode (Gnl, Var)
{
    GNL           Gnl;
    GNL_VAR       Var;
{
    GNL_NODE     VarNode;

    if (GnlVarTag (Var) != GnlTag (Gnl))
    {
        SetGnlVarTag (Var, GnlTag (Gnl));

        if (GnlCreateNodeForVar (Gnl, Var, &VarNode))
        {
            fprintf (stderr, "RMV-ERR: Can't create input node\n");
            return (NULL);
        }

        SetGnlVarDads (Var, (BLIST)VarNode);

        /*
        fprintf (stderr, "Created new Node for Var %s\n", GnlVarName (Var));
        */
    }
    else
        VarNode = (GNL_NODE)GnlVarDads (Var);

    return (VarNode);
}

/*-----*/
/* GetNonVarNode */
/*-----*/
GNL_NODE GetNonVarNode (Gnl, Var)
{
    GNL           Gnl;
    GNL_VAR       Var;
{
    GNL_VAR       RightVar;
    GNL_NODE     VarNode;
    int          Rank;

    while (1)
    {
        VarNode = GnlVarOnSetNode (Var);

        if (!VarNode)
        {

```

```

Rank = BListMemberOfList (GnlLocals (Gnl), (int)Var, IntIdentical);
BListDelInsert (GnlLocals (Gnl), Rank);

if (!(RightVar = GetBuffInVar (Gnl, Var)))
{
    fprintf (stderr, "RMV-ERR: variable %s doesn't have any support
!\n",
             GnlVarName (Var));
    return (NULL);
}

if (GnlVarDir (RightVar) == GNL_VAR_INPUT)
    return (CreateInputNode (Gnl, RightVar));

Var = RightVar;
continue;
}

if (GnlNodeOp (VarNode) == GNL_CONSTANTE)
{
    if (GnlVarTag (Var) != GnlTag (Gnl))
    {
        SetGnlVarTag (Var, GnlTag (Gnl));
        SetGnlVarDads (Var, (BLIST)VarNode);
        SetGnlConstant (Gnl, (int)GnlNodeSons (VarNode), Var);
    }
    else
        VarNode = (GNL_NODE)GnlVarDads (Var);

    return (VarNode);
}

if (GnlNodeOp (VarNode) == GNL_VARIABLE)
{
    RightVar = (GNL_VAR)GnlNodeSons (VarNode);

    if (GnlVarDir (RightVar) == GNL_VAR_INPUT)
    {
        if (GnlVarTag (RightVar) != GnlTag (Gnl))
        {
            SetGnlVarTag (RightVar, GnlTag (Gnl));
            SetGnlVarDads (RightVar, (BLIST)VarNode);
        }
        else
            VarNode = (GNL_NODE)GnlVarDads (RightVar);

        return (VarNode);
    }

    Var = RightVar;
    continue;
}

return (VarNode);
}
}

```

gnlec.c

```
/*-----*/
/* RemoveLocalSons */
/*-----*/
GNL_STATUS RemoveLocalSons (Gnl, Node)
    GNL        Gnl;
    GNL_NODE  Node;
{
    int      i;
    GNL_NODE SonI;
    GNL_VAR  VarI;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return (GNL_OK);

    SetGnlNodeTag (Node, GnlTag (Gnl));

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

        if (GnlNodeTag (SonI) == GnlTag (Gnl))
            continue;

        if (GnlNodeOp (SonI) == GNL_VARIABLE)
        {
            VarI = (GNL_VAR)GnlNodeSons (SonI);

            if (GnlVarDir (VarI) == GNL_VAR_INPUT)
            {
                if (GnlVarTag (VarI) != GnlTag (Gnl))
                {
                    SetGnlVarTag (VarI, GnlTag (Gnl));
                    SetGnlVarDads (VarI, (BLIST)SonI);
                }
                else
                    BListElt (GnlNodeSons (Node), i) = (int)GnlVarDads (VarI);

                continue;
            }
            else
            {
                if (!(SonI = GetNonVarNode (Gnl, VarI)))
                    return (GNL_BAD_GNL);
                BListElt (GnlNodeSons (Node), i) = (int)SonI;
            }
        }

        if ((GnlNodeOp (SonI) != GNL_VARIABLE) &&
            (GnlNodeOp (SonI) != GNL_CONSTANTE))
            if (RemoveLocalSons (Gnl, SonI))
                return (GNL_BAD_GNL);
    }

    return (GNL_OK);
}
/*-----*/
```

gnlec.c

```
/* RemoveLocals */  
/*-----*/  
/* This procedure removes all nodes corresponding to local variables */  
/* and assigns to the Dads field of the input Vars pointers to the */  
/* corresponding nodes. */  
/*-----*/  
GNL_STATUS RemoveLocals (Gnl, OutNodesList)  
    GNL      Gnl;  
    BLIST   OutNodesList;  
{  
    int      i;  
    GNL_VAR OutVarI;  
    GNL_NODE OutVarNode;  
  
    IncrGnlTag (Gnl);  
  
    for (i=0; i<BListSize (GnlVirtualOutputs (Gnl)); i++)  
    {  
        OutVarI = (GNL_VAR) BListElt (GnlVirtualOutputs (Gnl), i);  
  
        if (GnlVarDir (OutVarI) == GNL_VAR_INPUT)  
        {  
            if (!(OutVarNode = CreateInputNode (Gnl, OutVarI)))  
                return (GNL_MEMORY_FULL);  
            }  
        else  
            {  
                if (!(OutVarNode = GetNonVarNode (Gnl, OutVarI)))  
                    return (GNL_BAD_GNL);  
            }  
  
        if (BListAddElt (OutNodesList, (int)OutVarNode))  
            return (GNL_MEMORY_FULL);  
  
        if ((GnlNodeOp (OutVarNode) != GNL_VARIABLE) &&  
            (GnlNodeOp (OutVarNode) != GNL_CONSTANTE))  
            if (RemoveLocalSons (Gnl, OutVarNode))  
                return (GNL_BAD_GNL);  
        }  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* ReconnectNode */  
/*-----*/  
GNL_STATUS ReconnectNode (FNode, TNode, Invert)  
    GNL_NODE FNode;  
    GNL_NODE TNode;  
    int      *Invert;  
{  
    int      i;  
    int      NodesRank;  
    GNL_NODE DadI;  
    BLIST   Union;  
    GNL_NODE FromNode = FNode;  
    GNL_NODE ToNode = TNode;
```

```

BDD      NodesBdd;
BDD_PTR  NodesBddPtr;

*Invert = 0;

if (FNode == TNode)
    return (GNL_OK);

G_BridgeNodes++;

if (GnlNodeHeight (FNode) < GnlNodeHeight (TNode))
{
    FromNode = TNode;
    ToNode = FNode;
    *Invert = 1;
}

/*
fprintf (stderr, "Reconnecting: %s ---> %s\n",
        (GnlNodeVar (FromNode) ? GnlVarName (GnlNodeVar (FromNode)) : "No
var"),
        (GnlNodeVar (ToNode) ? GnlVarName (GnlNodeVar (ToNode)) : "No
var"));
 */

if (NodesBdd = GnlNodeBdd (FromNode))
{
    NodesBddPtr = GetBddPtr (NodesBdd);
    if (IsBddTypeINV (NodesBdd))
        SetGnlBddInvRefNode (NodesBddPtr, ToNode);
    else
        SetGnlBddPosRefNode (NodesBddPtr, ToNode);
}

for (i=0; i<BListSize (GnlNodeDads (FromNode)); i++)
{
    DadI = (GNL_NODE)BListElt (GnlNodeDads (FromNode), i);
    while (NodesRank = BListMemberOfList (GnlNodeSons (DadI),
                                         (int)FromNode, IntIdentical))
        BListElt (GnlNodeSons (DadI), NodesRank-1) = (int)ToNode;
}

if (BListCreateWithSize (BListSize (GnlNodeDads (FromNode)) +
                        BListSize (GnlNodeDads (ToNode)), &Union))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlNodeDads (ToNode)); i++)
    BListAddElt (Union, BListElt (GnlNodeDads (ToNode), i));
for (i=0; i<BListSize (GnlNodeDads (FromNode)); i++)
    BListAddElt (Union, BListElt (GnlNodeDads (FromNode), i));

BListQuickDelete (&GnlNodeDads (ToNode));
BListQuickDelete (&GnlNodeDads (FromNode));
SetGnlNodeDads (ToNode, Union);

if (GnlNodeBreakBddVar (FromNode))
    SetGnlNodeBreakBddVar (ToNode, GnlNodeBreakBddVar (FromNode));

```

```

    while (NodesRank = BListMemberOfList (G_OutputNodes1, (int)FromNode,
IntIdentical))
        BListElt (G_OutputNodes1, NodesRank-1) = (int)ToNode;
    while (NodesRank = BListMemberOfList (G_OutputNodes2, (int)FromNode,
IntIdentical))
        BListElt (G_OutputNodes2, NodesRank-1) = (int)ToNode;

    return (GNL_OK);
}

/*-----*/
/* ReconnectInvNode */
/*-----*/
GNL_STATUS ReconnectInvNode (Gnl, FNode, TNode, Invert)
    GNL          Gnl;
    GNL_NODE     FNode;
    GNL_NODE     TNode;
    int          *Invert;
{
    int          i = 0;
    int          Inv;
    int          NotHeight;
    GNL_NODE    FromNode = FNode;
    GNL_NODE    ToNode = TNode;
    GNL_NODE    NotNode;
    BLIST       Sons;
    void        *ptr;

    *Invert = -1;

    if ((GnlNodeOp (FNode) != GNL_NOT) &&
        (GnlNodeOp (TNode) != GNL_NOT))
    {
        *Invert = 0;

        if (GnlNodeHeight (FNode) < GnlNodeHeight (TNode))
        {
            FromNode = TNode;
            ToNode = FNode;
            *Invert = 1;
        }

        while (i < BListSize (GnlNodeDads (ToNode)))
            if (GnlNodeOp (NotNode = (GNL_NODE)BListElt (GnlNodeDads (ToNode),
i++)) == GNL_NOT)
                break;

        if (i == BListSize (GnlNodeDads (ToNode)))
        {
            G_InvBridgeNodes++;

            if (GnlCreateNodeNot (Gnl, ToNode, &NotNode))
                return (GNL_MEMORY_FULL);

            if ((Ptr = calloc (1, sizeof (ADD_GNL_NODE_REC))) == NULL)
                return (GNL_MEMORY_FULL);
        }
    }
}

```

```

    SetGnlNodeHook (NotNode, Ptr);
    SetGnlNodeHeight (NotNode, GnlNodeHeight (ToNode) + 1);
    if (AllocateHookPtrs (NotNode))
        return (GNL_MEMORY_FULL);

    if (BListAddEltWCheck (&GnlNodeDads (ToNode), (int)NotNode))
        return (GNL_MEMORY_FULL);
    }

    SetGnlNodeBdd (NotNode, UseBdd (GnlNodeBdd (FromNode)));
    SetGnlNodeBreakBddVar (NotNode, GnlNodeBreakBddVar (FromNode));

    NotHeight = GnlNodeHeight (NotNode);
    SetGnlNodeHeight (NotNode, 0);
    if (ReconnectNode (FromNode, NotNode, &Inv))
        return (GNL_MEMORY_FULL);
    SetGnlNodeHeight (NotNode, NotHeight);

    return (GNL_OK);
}

return (GNL_OK);
}

/*-----*/
/* ReconnectToORNode */
/*-----*/
/*
GNL_STATUS ReconnectToORNode (RefGnl, FromNode, ToNode)
    GNL_RefGnl;
    GNL_NODE FromNode;
    GNL_NODE ToNode;
{
    int i;
    int NodesRank;
    int MaxHeight;
    GNL_NODE OrNode;
    void *Hook;

    G_Impl++;

    if (GnlCreateNode (RefGnl, GNL_OR, &OrNode))
        return (GNL_MEMORY_FULL);

    SetGnlNodeDads (OrNode, GnlNodeDads (ToNode));

    if (BListCreateWithSize (2, &GnlNodeSons (OrNode)))
        return (GNL_MEMORY_FULL);

    BListAddElt (GnlNodeSons (OrNode), (int)FromNode);
    BListAddElt (GnlNodeSons (OrNode), (int)ToNode);

    if (BListAddEltWCheck (&GnlNodeDads (FromNode), (int)OrNode))
        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (1, &GnlNodeDads (ToNode)))
        return (GNL_MEMORY_FULL);

```

```

BListAddElt (GnlNodeDads (ToNode), (int)OrNode);

if ((Hook = calloc (1, sizeof (ADD_GNL_NODE_REC))) == NULL)
    return (GNL_MEMORY_FULL);

SetGnlNodeHook (OrNode, Hook);

MaxHeight = (GnlNodeHeight (FromNode) > GnlNodeHeight (ToNode) ?
             GnlNodeHeight (FromNode) : GnlNodeHeight (ToNode));
SetGnlNodeHeight (OrNode, MaxHeight + 1);

if (AllocateHookPtrs (OrNode))
    return (GNL_MEMORY_FULL);

while (NodesRank = BListMemberOfList (G_OutputNodes1, (int)ToNode,
IntIdentical))
    BListElt (G_OutputNodes1, NodesRank-1) = (int)OrNode;
    while (NodesRank = BListMemberOfList (G_OutputNodes2, (int)ToNode,
IntIdentical))
        BListElt (G_OutputNodes2, NodesRank-1) = (int)OrNode;

    return (GNL_OK);
}
*/
/*-----*/
/* BindPrimaryInputs */
/*-----*/
/* ASSUMPTION!: the two Gnl must have the same number of primary inputs */
/*-----*/
GNL_STATUS BindPrimaryInputs (Gnl1, Gnl2)
    GNL      Gnl1;
    GNL      Gnl2;
{
    int          i;
    GNL_VAR     Var1;
    GNL_VAR     Var2;
    GNL_NODE   Node1;
    GNL_NODE   Node2;
    int          Invert;

    for (i=0; i<BListSize (GnlRelevantInputs (Gnl1)); i++)
    {
        /* We reconnect the relevant primary inputs' nodes of Gnl2 to      */
        /* the corresponding relevant primary inputs' nodes of Gnl1      */
        Var1 = (GNL_VAR)BListElt (GnlRelevantInputs (Gnl1), i);
        Var2 = (GNL_VAR)BListElt (GnlRelevantInputs (Gnl2), i);
        Node1 = (GNL_NODE)GnlVarDads (Var1);
        Node2 = (GNL_NODE)GnlVarDads (Var2);
        if (!Node1)
        {
            BListElt (GnlRelevantInputs (Gnl1), i) = (int)Var2;
            continue;
        }
        if (!Node2)
            continue;
    }
}

```

gnlec.c

```
if (ReconnectNode (Node2, Node1, &Invert))
    return (GNL_MEMORY_FULL) ;
}

for (i=0; i<2; i++)
    if ((Var1 = GnlConstant (Gnl1, i)) && (Var2 = GnlConstant (Gnl2, i)))
    {
        Node1 = (GNL_NODE) GnlVarDads (Var1);
        Node2 = (GNL_NODE) GnlVarDads (Var2);
        if (ReconnectNode (Node2, Node1, &Invert))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
#define MAX_LAYER  97
/*-----*/
/* INDEX */
/*-----*/
int INDEX (Node)
    GNL_NODE  Node;
{
    int      i;
    int      Value;

    if (GnlNodeValue (Node) == -1)
    {
        Value = (int) GnlNodeOp (Node) + BListSize (GnlNodeSons (Node));
        for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
            Value += GnlNodeValue ((GNL_NODE) BListElt (GnlNodeSons (Node), i));
        Value = Value % MAX_LAYER;
        SetGnlNodeValue (Node, Value);
    }
    return (GnlNodeValue (Node));
}

/*-----*/
/* AddDads */
/*-----*/
/* Adds the node Node in the hash table "Layer" if it does not exist */
/*-----*/
GNL_STATUS AddDads (Gnl, AllLayer, Node)
    GNL          Gnl;
    BLIST        *AllLayer;
    GNL_NODE     Node;
{
    int           Key;
    int           i;
    GNL_NODE     DadI;
    BLIST        Layer;

    if (!Node || !GnlNodeHeight (Node))
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeDads (Node)); i++)
```

```

{
/* Dad's tag means that he has been added previously so we */      */
/* do not add it . */      */
DadI = (GNL_NODE)BListElt (GnlNodeDads (Node), i);
if (GnlNodeTag (DadI) == GnlTag (Gnl))
continue;

SetGnlNodeTag (DadI, GnlTag (Gnl));

Key = INDEX (DadI);

/* Selecting the Layer of the DadI corresponding to its Height */
Layer = AllLayer[GnlNodeHeight (DadI) - 1];
if (BListAddEltWCheck ((BLIST*)&BListElt (Layer, Key), (int)DadI))
return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----*/
/* ComputeDadsOnNodes */
/*-----*/
int ComputeDadsOnNodes (Gnl, Node)
    GNL          Gnl;
    GNL_NODE Node;
{
    int           i;
    GNL_NODE NodeSonI;
    BLIST        NodeSonIDads;
    BLIST        VarSonIDads;
    GNL_VAR     VarSonI;

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE) ||
        (GnlNodeTag (Node) == GnlTag (Gnl)))
    return (0);

    SetGnlNodeTag (Node, GnlTag (Gnl));

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeSonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        NodeSonIDads = GnlNodeDads (NodeSonI);
        if (BListAddEltWCheck (&NodeSonIDads, (int)Node))
        return (1);
        SetGnlNodeDads (NodeSonI, NodeSonIDads);

        if (ComputeDadsOnNodes (Gnl, NodeSonI))
        return (1);
    }

    return (0);
}

```

gnlec.c

```
-----*/
/* ComputeDadsInGnl */
-----*/
static int ComputeDadsInGnl (Gnl1, Gnl2)
    GNL          Gnl1;
    GNL          Gnl2;
{
    int           i;
    GNL_NODE OutI;

    MakeNewTag (Gnl1, Gnl2);

    for (i=0; i<BListSize (G_OutputNodes1); i++)
    {
        OutI = (GNL_NODE)BlistElt (G_OutputNodes1, i);
        if (ComputeDadsOnNodes (Gnl1, OutI))
            return (1);
    }

    for (i=0; i<BListSize (G_OutputNodes2); i++)
    {
        OutI = (GNL_NODE)BlistElt (G_OutputNodes2, i);
        if (ComputeDadsOnNodes (Gnl2, OutI))
            return (1);
    }

    return (0);
}

-----*/
/* ComputeNodesHeight */
-----*/
void ComputeNodesHeight (Gnl, Node)
    GNL          Gnl;
    GNL_NODE Node;
{
    GNL_NODE     SonI;
    int          i;
    int          Height;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return;

    SetGnlNodeTag (Node, GnlTag (Gnl));

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
    {
        SetGnlNodeHeight (Node, 1);
        return;
    }

    SetGnlNodeHeight (Node, 0);
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BlistElt (GnlNodeSons (Node), i);
```

```

        ComputeNodesHeight (Gnl, SonI);
        if ((Height = GnlNodeHeight (SonI) + 1) > GnlNodeHeight (Node))
            SetGnlNodeHeight (Node, Height);
    }
}

/*-----
/* ComputeGnlHeights
/*-----
/* The Height of a var is 1 for primary inputs and
/* the Height of the corresponding node otherwise
/*-----
void ComputeGnlHeights (Gnl1, Gnl2, MaxHeight)
{
    GNL          Gnl1;
    GNL          Gnl2;
    int          *MaxHeight;
{
    int          i;
    GNL_NODE     OutNodeI;

    *MaxHeight = 0;

    MakeNewTag (Gnl1, Gnl2);

    for (i=0; i<BListSize (G_OutputNodes1); i++)
    {
        OutNodeI = (GNL_NODE)BListElt (G_OutputNodes1, i);
        ComputeNodesHeight (Gnl1, OutNodeI);
        if (*MaxHeight < GnlNodeHeight (OutNodeI))
            *MaxHeight = GnlNodeHeight (OutNodeI);

        OutNodeI = (GNL_NODE)BListElt (G_OutputNodes2, i);
        ComputeNodesHeight (Gnl2, OutNodeI);
        if (*MaxHeight < GnlNodeHeight (OutNodeI))
            *MaxHeight = GnlNodeHeight (OutNodeI);
    }
}

/*-----
/* CreateAllLayers
/*-----
int CreateAllLayers (MaxHeight, AllLayers)
{
    int          MaxHeight;
    BLIST       **AllLayers;
{
    int          i;

    if ((*AllLayers = (BLIST*)calloc (MaxHeight, sizeof (BLIST))) == NULL)
        return (1);

    for (i=0; i<MaxHeight; i++)
        if (BlistCreateWithSize (MAX_LAYER, &((*AllLayers) [i])))
            return (1);

    return (0);
}

```

```

/*-----*/
/* FreeAllLayers */
/*-----*/
void FreeAllLayers (MaxHeight, AllLayers)
    int      MaxHeight;
    BLIST   *AllLayers;
{
    int      i;

    for (i=0; i<MaxHeight; i++)
        BListDelete (&(AllLayers[i]), BListQuickDelete);

    free (AllLayers);
}

/*-----*/
/* EquivStructuralNode */
/*-----*/
int EquivStructuralNode (Node1, Node2)
    GNL_NODE   Node1;
    GNL_NODE   Node2;
{
    int      i;
    int      j;
    int      NumSons1;
    int      NumSons2;
    GNL_NODE  Son1;
    GNL_NODE  Son2;
    int      Found;

    if (Node1 == Node2)
        return (1);

    if (GnlNodeOp (Node1) != GnlNodeOp (Node2))
        return (0);

    if ((NumSons1 = BListSize (GnlNodeSons (Node1))) !=
        (NumSons2 = BListSize (GnlNodeSons (Node2))))
        return (0);

    for (i=0; i<NumSons1; i++)
    {
        Son1 = (GNL_NODE)BListElt (GnlNodeSons (Node1), i);
        Found = 0;
        for (j=0; j<NumSons2; j++)
        {
            Son2 = (GNL_NODE)BListElt (GnlNodeSons (Node2), j);
            if (Son1 == Son2)
            {
                Found = 1;
                break;
            }
        }
        if (!Found)
            return (0);
    }
}

```

```

    }

    return (1);
}

/*-----
/* MatchInternal                               */
/*-----*/
GNL_STATUS MatchInternal (Gn11, Gn12, ListNodes, AllLayers)
    GNL          Gn11;
    GNL          Gn12;
    BLIST        ListNodes;
    BLIST        *AllLayers;
{
    int          i;
    int          j;
    GNL_NODE    Node1;
    GNL_NODE    Node2;
    int          Invert = 0;

    for (i=0; i<BListSize (ListNodes); i++)
        for (j=i+1; j<BListSize (ListNodes); j++)
    {
        Node1 = (GNL_NODE)BListElt (ListNodes, i);
        Node2 = (GNL_NODE)BListElt (ListNodes, j);
        if (EquivStructuralNode (Node1, Node2))
        {
            if (AddDads (Gn11, AllLayers, Node1))
                return (GNL_MEMORY_FULL);

            if (Node1 != Node2)
            {
                if (AddDads (Gn11, AllLayers, Node2))
                    return (GNL_MEMORY_FULL);

                if (ReconnectNode (Node2, Node1, &Invert))
                    return (GNL_MEMORY_FULL);
            }

            if (Invert)
                BListElt (ListNodes, i) = (int)Node2;

            BListDelInsert (ListNodes, j+1);
            j--;
        }
    }
    return (GNL_OK);
}

/*-----
/* MatchNodes                                */
/*-----*/
/* This function identifies structurally equivalent Nodes in the layers */
/* Layer1 and Layer2 and removes them from the layers.                      */
/*-----*/
GNL_STATUS MatchNodes (Gn11, Gn12, AllLayer, Height)
    GNL          Gn11;

```

```

    GNL          Gnl2;
    BLIST      *AllLayer;
    int         Height;
{
    int         Key;
    BLIST      Layer;
    BLIST      KeysLayer;

    Layer = AllLayer[Height-1];

    for (Key=0; Key<MAX_LAYER; Key++)
    {
        if (KeysLayer = (BLIST)BListElt (Layer, Key))
        if (MatchInternal (Gnl1, Gnl2, KeysLayer, AllLayer))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* EquivCheckGnlWithStructural */
/*-----*/
/* This function connect GNL_NODE and GNL_VAR objects of the two */
/* netlists Gnl1 and Gnl2 by filling their fields BindedNode and */
/* BindedVar for cross referencing. Two elements are connected if they */
/* are structurally equivalent. To do so we start from the leaves of */
/* each netlist by connecting first the primary inputs, then we use a */
/* bottom-up merging. The function returns 1 if any problem occurs */
/* (Memory full, ...) and 0 if everything was OK. It returns also the */
/* number of different outputs which were not structurally merged */
/* (var. NbFalse) and the list of these outputs for both netlists */
/* (OutGnl1 and OutGnl2). */
/*-----*/
GNL_STATUS EquivCheckGnlWithStructural (Gnl1, Gnl2, OutMayFalse)
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST      *OutMayFalse;
{
    int         i;
    int         NbIn;
    int         NbOut;
    int         NbFalse;
    GNL_NODE Out1;
    GNL_NODE Out2;
    int         Height;
    int         MaxHeight;
    BLIST      *AllLayers;

    NbIn = BListSize (GnlRelevantInputs (Gnl1));
    NbOut = BListSize (GnlVirtualOutputs (Gnl1));

    /* We compute the Height of each Node in the two Netlists. */
    ComputeGnlHeights (Gnl1, Gnl2, &MaxHeight); */

```

```

/* We create the set of Layers that we will use to make the matching */
if (CreateAllLayers (MaxHeight, &AllLayers))
    return (GNL_MEMORY_FULL);

MakeNewTag (Gnl1, Gnl2);

/* The primary inputs' nodes belong to the Layer 1. */
for (i=0; i<NbIn; i++)
{
    if (AddDads (Gnl1, AllLayers,
        (GNL_NODE)GnlVarDads ((GNL_VAR)BListElt (GnlRelevantInputs
(Gnl1), i))))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<2; i++)
{
    if (GnlConstant (Gnl1, i))
        if (AddDads (Gnl1, AllLayers, (GNL_NODE)GnlVarDads (GnlConstant (Gnl1,
i))))
            return (GNL_MEMORY_FULL);
}

/* We start from Layer 2. */
for (Height = 2; Height <= MaxHeight; Height++)
    if (MatchNodes (Gnl1, Gnl2, AllLayers, Height))
        return (GNL_MEMORY_FULL);

NbFalse = 0;
for (i=0; i<NbOut; i++)
{
    Out1 = (GNL_NODE)BListElt (G_OutputNodes1, i);
    Out2 = (GNL_NODE)BListElt (G_OutputNodes2, i);

    if (Out1 != Out2)
        NbFalse++;
}

if (NbFalse == 0)
    return (0);

if (BListCreateWithSize (NbFalse, OutMayFalse))
    return (GNL_MEMORY_FULL);

for (i=0; i<NbOut; i++)
{
    Out1 = (GNL_NODE)BListElt (G_OutputNodes1, i);
    Out2 = (GNL_NODE)BListElt (G_OutputNodes2, i);

    if (Out1 != Out2)
    {
        if (BListAddElt (*OutMayFalse, i))
            return (GNL_MEMORY_FULL);
    }
}
*/

```

```

    else
    {
        fprintf (stderr, "\n <%s> == <%s>\n",
                 GnlVarName ((GNL_VAR)BListElt (G_Outputs1, i)),
                 GnlVarName ((GNL_VAR)BListElt (G_Outputs2, i)));
    }
    */
}

FreeAllLayers (MaxHeight, AllLayers);

return (GNL_OK);
}

/* -----
/* GnlBuildBddOnVarInput
/* -----
BDD_STATUS GnlBuildBddOnVarInput (Gnl)
    GNL          Gnl;
{
    int          i;
    GNL_VAR     VarI;
    BDD_STATUS   Status;
    BDD_VAR     VarBdd;

    for (i=0; i<BListSize (GnlRelevantInputs (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlRelevantInputs (Gnl), i);
        if (Status = CreateBddVar (GnlVarName (VarI), &VarBdd))
            return (Status);

        SetGnlVarBdd (VarI, UseBdd (BddVarBdd (VarBdd)));
    }

    return (GNL_OK);
}

/* -----
/* ComputeBddOnNode
/* -----
/* Compute BDD of node 'Node' and stores it in the field Node->Bdd. */
/* -----
BDD_STATUS ComputeBddOnNode (Node, Mode, OverLimit, BddRes)
    GNL_NODE   Node;
    int         Mode;
    int         *OverLimit;
    BDD         *BddRes;
{
    int          i;
    GNL_NODE   SonI;
    BDD         SonIBdd;
    BDD_STATUS  Status;
    BDD         NewBdd;
    BDD         BddTemp;
    int         NbNodes;
}

```

```

*OverLimit = 0;

#ifndef PRINT_BDD_BUILD
    fprintf (stderr, "Began to compute BDD for node %x[%s]\n",
            Node, (GnlNodeVar (Node) ? GnlVarName (GnlNodeVar (Node)) : "No
var"));
#endif

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

    if ((GnlNodeOp (SonI) == GNL_CONSTANTE) ||
        (GnlNodeOp (SonI) == GNL_VARIABLE) ||
        (Mode != BREAK_MODE))
        SonIBdd = GnlNodeBdd (SonI);
    else
        SonIBdd = BddVarBdd (GnlNodeBreakBddVar (SonI));

    if (i == 0)                                /* First son: we set 'NewBdd' */
    {
        NewBdd = SonIBdd;
        if (GnlNodeOp (Node) != GNL_NOT)
            continue;
        if (bdd_not (NewBdd, BddRes))
            return (GNL_MEMORY_FULL);
        return (BDD_OK);
    }

    switch (GnlNodeOp (Node))
    {
        case GNL_AND:
        if (bdd_and (SonIBdd, NewBdd, &BddTemp))
            return (GNL_MEMORY_FULL);
        break;

        case GNL_OR:
        if (bdd_or (SonIBdd, NewBdd, &BddTemp))
            return (GNL_MEMORY_FULL);
        break;

        default:
        fprintf (stderr, "ERROR: unknown operator\n");
        exit (1);
    }

    NewBdd = BddTemp;

    if (Mode == REGULAR_MODE)
    {
        BddCountNodes (NewBdd, &NbNodes);

        if (NbNodes > (G_GnlMaxBddNode * (i == (BListSize (GnlNodeSons (Node))
- 1) ?
                                            1 : BListSize (GnlNodeSons (Node))))) )
    {
        *OverLimit = 1;
    }
}

```

```

        return (BDD_OK);
    }
}
}

#endif PRINT_BDD_BUILD
fprintf (stderr, "Finished to compute BDD for node %x[%s]\n",
         Node, (GnlNodeVar (Node)) ? GnlVarName (GnlNodeVar (Node)) : "No
var"));
#endif

*BddRes = NewBdd;
return (BDD_OK);
}

/* -----
/* GnlNodeUpdateBddRef
/* -----
GNL_STATUS GnlNodeUpdateBddRef (RefGnl, Mode, Node, Layers)
{
    GNL                      RefGnl;
    int                       Mode;
    GNL_NODE                  Node;
    BLIST                     *Layers;

    BDD                      NodesBdd;
    BDD_PTR                  BddResPtr;
    GNL_NODE                  RefNode;
    BDD_REF_INFO              BddRefInfo;
    int                       Invert;
    int                       Rank;

    NodesBdd = (Mode == BREAK_MODE ?
                 BddVarBdd (GnlNodeBreakBddVar (Node)) : GnlNodeBdd (Node));
    BddResPtr = GetBddPtr (NodesBdd);

    /* we create the structure info if does not exist for 'BddResPtr' */
    if (!GetBddPtrHook (BddResPtr))
    {
        if (GnlCreateBddRefInfo (&BddRefInfo))
            return (GNL_MEMORY_FULL);
        SetBddPtrHook (BddResPtr, (Uint32)BddRefInfo);
        if (IsBddTypeINV (NodesBdd))
            SetGnlBddInvRefNode (BddResPtr, Node);
        else
            SetGnlBddPosRefNode (BddResPtr, Node);
        return (GNL_OK);
    }

    if (IsBddTypeINV (NodesBdd) && (GnlBddInvRefNode (BddResPtr) == NULL))
    {
        SetGnlBddInvRefNode (BddResPtr, Node);
        if (GnlBddPosRefNode (BddResPtr))
        {
            if (ReconnectInvNode (RefGnl, Node, GnlBddPosRefNode (BddResPtr),
&Invert))
                return (GNL_MEMORY_FULL);
        }
    }
}

```

```

    if (Invert == -1)
        return (GNL_OK);

    if (Layers)
    {
        RefNode = (Invert ? GnlBddPosRefNode (BddResPtr) : Node);

        Rank = BListMemberOfList (Layers[GnlNodeHeight (RefNode)-1],
(int)RefNode);
        BListDelShift (Layers[GnlNodeHeight (RefNode)-1], Rank);
    }
    return (GNL_OK);
}

if (!IsBddTypeINV (NodesBdd) && (GnlBddPosRefNode (BddResPtr) == NULL))
{
    SetGnlBddPosRefNode (BddResPtr, Node);
    if (GnlBddInvRefNode (BddResPtr))
    {
        if (ReconnectInvNode (RefGnl, Node, GnlBddInvRefNode (BddResPtr),
&Invert))
            return (GNL_MEMORY_FULL);

        if (Invert == -1)
            return (GNL_OK);

        if (Layers)
        {
            RefNode = (Invert ? GnlBddInvRefNode (BddResPtr) : Node);

            Rank = BListMemberOfList (Layers[GnlNodeHeight (RefNode)-1],
(int)RefNode);
            BListDelShift (Layers[GnlNodeHeight (RefNode)-1], Rank);
        }
        return (GNL_OK);
    }
}

/* There is already a Ref node. We reconnect Node to the ref. node */
if (IsBddTypeINV (NodesBdd))
    RefNode = GnlBddInvRefNode (BddResPtr);
else
    RefNode = GnlBddPosRefNode (BddResPtr);

if (ReconnectNode (Node, RefNode, &Invert))
    return (GNL_MEMORY_FULL);

if (Layers)
{
    RefNode = (Invert ? RefNode : Node);

    Rank = BListMemberOfList (Layers[GnlNodeHeight (RefNode)-1],
(int)RefNode);
    BListDelShift (Layers[GnlNodeHeight (RefNode)-1], Rank);
}

```

```

    return (GNL_OK);
}

/* -----
/* GnlBuildBddFromSons
/* -----
GNL_STATUS GnlBuildBddFromSons (RefGnl, Node)
    GNL             RefGnl;
    GNL_NODE        Node;
{
    int             i;
    GNL_NODE        SonI;
    BDD             BddRes;
    int             BddOverLimit = 0;
    BDD_VAR         BddVar;
    char            *NewStr;

    if (ComputeBddOnNode (Node, REGULAR_MODE, &BddOverLimit, &BddRes))
        return (GNL_MEMORY_FULL);

    if (BddOverLimit)
    {
        /*
        printf ("Node = %x, Var = %s\n", Node,
               (GnlNodeVar (Node) ? GnlVarName (GnlNodeVar (Node)) : "No var"));
        printf ("  CutSons:\n");
        */

        /* The Max Bdd node limit has been reached and we need to          */
        /* create break bdd var for the sons of node so we can build       */
        /* its associated bdd function.                                     */
        for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        {
            SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

            /* Actually the node 'SonI' has already a break bdd var      */
            if ((GnlNodeOp (SonI) == GNL_CONSTANTE) ||
                (GnlNodeOp (SonI) == GNL_VARIABLE) ||
                GnlNodeBreakBddVar (SonI))
                continue;

            if (GnlStrAppendIntCopy ("X", GLOB_BDD_WS->UniqueId+1, &NewStr))
                return (GNL_MEMORY_FULL);
            if (CreateBddVar (NewStr, &BddVar))
                return (GNL_MEMORY_FULL);
            SetGnlNodeBreakBddVar (SonI, BddVar);

            if (GnlNodeUpdateBddRef (RefGnl, BREAK_MODE, SonI, NULL))
                return (GNL_MEMORY_FULL);

            G_NbCutVar++;
            /*
            printf ("%d. Son[%d] = %x, Var = %s, CutVar = %s\n", G_NbCutVar, i,
SonI,
               (GnlNodeVar (SonI) ? GnlVarName (GnlNodeVar (SonI)) : "No var"),
NewStr);

```

```

        */

    }

    /* Now we rebuild the new Bdd by taking into account           */
    /* the break bdd var                                         */
    if (ComputeBddOnNode (Node, BREAK_MODE, &BddOverLimit, &BddRes))
        return (GNL_MEMORY_FULL);
    }

SetGnlNodeBdd (Node, UseBdd (BddRes));

if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/* -----
/* GnlNodeBuildBddFct                                     */
/* ----- */
/* This procedure builds the Bdd function of the function corresponding */
/* to node 'Node'. The Bdd size cannot exceeds the size specified in   */
/* 'G_GnlMaxBddNode'. If this is the case a break var is inserted at the*/
/* sons of the node and the bdd of the node is simply the bdd apply on   */
/* this node with the break vars.                                     */
/* ----- */
GNL_STATUS GnlNodeBuildBddFct (RefGnl, Node)
    GNL             RefGnl;
    GNL_NODE        Node;
{
    int             i;
    GNL_NODE        SonI;

    /* The Node has actually already a Bdd function assosciated.          */
    if (GnlNodeBdd (Node))
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        if (GnlNodeSons (Node))
            SetGnlNodeBdd (Node, bdd_one());
        else
            SetGnlNodeBdd (Node, bdd_zero());

        if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
            return (GNL_MEMORY_FULL);

        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        SetGnlNodeBdd (Node, UseBdd (GnlVarBdd (GnlNodeVar (Node))));

        if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
            return (GNL_MEMORY_FULL);
    }
}

```

```

        return (GNL_OK);
    }

/* First we build the Bdd functions of each son. */ 
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlNodeBuildBddFct (RefGnl, SonI))
        return (GNL_MEMORY_FULL);
}

/* Now we compute the Bdd function of 'Node' by applying the */ 
/* corresponding operator of 'GnlNodeOp'. */ 

return (GnlBuildBddFromSons (RefGnl, Node));
}

/* ----- */ 
/* GnlBuildBddFct */ 
/* ----- */ 
GNL_STATUS GnlBuildBddFct (Gnl1, Gnl2, OutsGnl)
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST        OutsGnl;
{
    int           i;
    int           Ind;
    BDD_WS      Ws;
    GNL_NODE NodeI;

    if (InitBddWorkSpaceInvOrder (BListSize (GnlRelevantInputs (Gnl1))+1,
&Ws))
        return (GNL_MEMORY_FULL);

    if (GnlBuildBddOnVarInput (Gnl1))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (OutsGnl); i++)
    {
        Ind = BListElt (OutsGnl, i);
        NodeI = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
        if (GnlNodeBuildBddFct (Gnl1, NodeI))
            return (GNL_MEMORY_FULL);
    }

    for (i=0; i<BListSize (OutsGnl); i++)
    {
        Ind = BListElt (OutsGnl, i);
        NodeI = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
        if (GnlNodeBuildBddFct (Gnl2, NodeI))
            return (GNL_MEMORY_FULL);
    }

    G_Tag++;

    for (i=0; i<BListSize (OutsGnl); i++)
    {

```

```

    Ind = BListElt (OutsGnl, i);

    NodeI = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
    GnlModifyEcNodeInfo (NodeI);

    NodeI = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
    GnlModifyEcNodeInfo (NodeI);
}

FreeBddWorkSpace (Ws);

return (GNL_OK);
}

#ifndef BFS_BDD_PHASE
/* -----
/* MarkFanIn
/* -----
void MarkFanIn (Gnl, Node)
    GNL          Gnl;
    GNL_NODE     Node;
{
    int          i;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return;

    SetGnlNodeTag (Node, GnlTag (Gnl));

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        MarkFanIn (Gnl, (GNL_NODE)BListElt (GnlNodeSons (Node), i));
}
/* -----
/* UpdateFanOutHeight
/* -----
void UpdateFanOutHeight (RefGnl, Node)
    GNL          RefGnl;
    GNL_NODE     Node;
{
    int          i;
    BLIST        Dads;
    GNL_NODE     DadI;

    if (!(Dads = GnlNodeDads (Node)))
        return;

    for (i=0; i<BListSize (Dads); i++)
    {
        DadI = (GNL_NODE)BListElt (Dads, i);

        if (GnlNodeTag (DadI) < GnlTag (RefGnl))
            continue;
    }
}

```

```

        if (GnlNodeHeight (DadI) < GnlNodeHeight (Node) + 1)
        {
            SetGnlNodeHeight (DadI, GnlNodeHeight (Node) + 1);
            UpdateFanOutHeight (DadI);
        }
    }

/*
 * BuildBddForLayer
 */
GNL_STATUS BuildBddForLayer (Layers, Height)
    BLIST      Layers;
    int       Height;
{
}

/*
 * GnlLocalBddPhase
 */
GNL_STATUS GnlLocalBddPhase (Gnl1, Gnl2, OutsMayFalse)
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST        OutsMayFalse;
{
    int          i;
    int          j;
    int          k;
    int          Ind;
    int          HeightI;
    int          MaxHeight = 0;
    int          Repeat;
    BDD_WS      Ws;
    GNL_VAR     InVarI;
    GNL_NODE    InNodeI;
    GNL_NODE    NodeJ;
    GNL_NODE    DadK;
    GNL_STATUS  Status;
    BDD_VAR     VarBdd;
    BLIST        *Layers;

    MakeNewTag (Gnl1, Gnl2);
    MakeNewTagBdd (Gnl1, Gnl2);

    for (i=0; i<BListSize (OutsMayFalse); i++)
    {
        Ind = BListElt (OutsMayFalse, i);
        MarkFanIn (Gnl1, (GNL_NODE)BListElt (G_OutputNodes1, Ind));
        MarkFanIn (Gnl2, (GNL_NODE)BListElt (G_OutputNodes2, Ind));

        if ((HeightI = GnlNodeHeight ((GNL_NODE)BListElt (G_OutputNodes1,
Ind))) > MaxHeight)
            MaxHeight = HeightI;
    }
}

```

```

        if ((HeightI = GnlNodeHeight ((GNL_NODE)BListElt (G_OutputNodes2,
Ind))) > MaxHeight)
            MaxHeight = HeightI;
    }

    if (!(Layers = (BLIST *)calloc (MaxHeight, sizeof (BLIST))))
        return (GNL_MEMORY_FULL);

    for (i=0; i<MaxHeight; i++)
        if (BlistCreateWithSize (BListSize (GnlRelevantInputs (Gnl1)) + 2,
&Layers[i]))
            return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlRelevantInputs (Gnl1)); i++)
    {
        InVarI = (GNL_VAR)BListElt (GnlRelevantInputs (Gnl1), i);

        if ((InNodeI = (GNL_NODE)GnlVarDads (InVarI)) &&
            (GnlNodeTag (InNodeI) == GnlTag (Gnl1)))
            BListAddElt (Layers[0], (int)InNodeI);
    }

    if (InitBddWorkSpaceInvOrder (BListSize (Layers[0]) + 1, &Ws))
        return (GNL_MEMORY_FULL);

    G_WsNum = 1;

    for (i=0; i<2; i++)
        if ((InVarI = GnlConstant (Gnl1, i)) && (InNodeI = (GNL_NODE)GnlVarDads
(InVarI)) &&
            (GnlNodeTag (InNodeI) == GnlTag (Gnl1)))
            BListAddElt (Layers[0], (int)InNodeI);

    for (i=0; i<BListSize (Layers[0]); i++)
    {
        InNodeI = (GNL_NODE)BListElt (Layers[0], i);
        InVarI = GnlNodeVar (InNodeI);

        if (GnlNodeOp (InNodeI) == GNL_CONSTANTE)
            SetGnlNodeBdd (InNodeI, (GnlNodeSons (InNodeI) ? bdd_one () : bdd_zero
()));
        else
        {
            if (Status = CreateBddVar (GnlVarName (InVarI), &VarBdd))
                return (Status);

            SetGnlVarBdd (InVarI, UseBdd (BddVarBdd (VarBdd)));
            SetGnlNodeBdd (InNodeI, UseBdd (BddVarBdd (VarBdd)));
        }
    }

    if (GnlNodeUpdateBddRef (Gnl1, REGULAR_MODE, InNodeI, Layers))
        return (GNL_MEMORY_FULL);
    }

    Repeat = 0;

    for (i=1; i<MaxHeight; i++)

```

```

{
    if (!Repeat)
    {
        for (j=0; j<BListSize (Layers[i]); j++)
        {
            NodeJ = (GNL_NODE)BListElt (Layers[i], j);

            for (k=0; k<BListSize (GnlNodeDads (NodeJ)); k++)
            {
                DadK = (GNL_NODE)BListElt (GnlNodeDads (NodeJ), k);

                if ((GnlNodeTag (DadK) == GnlTag (Gnl1)) &&
                    (GnlNodeTagBdd (DadK) < GnlTagBdd (Gnl1)))
                {
                    SetGnlNodeTagBdd (DadK, GnlTagBdd (Gnl1));
                    if (BListAddElt (Layers[i+1], (int)DadK))
                        return (GNL_MEMORY_FULL);
                }
            }
        }

        if (BuildBddForLayer (Layers, i))
        {
            G_WsNum++;

            G_Tag++;

            for (j=0; j<BListSize (Layers[i]); j++)
            {
                NodeJ = (GNL_NODE)BListElt (Layers[i], j);
                GnlFreeNodesBddRefInfo (NodeJ);
            }

            FreeBddWorkSpace (Ws);

            if (InitBddWorkSpaceInvOrder (100, &Ws))
                return (GNL_MEMORY_FULL);

            i--;
            Repeat = 1;
            continue;
        }

        Repeat = 0;
    }

    for (i=0; i<MaxHeight; i++)
        BListQuickDelete (&Layers[i]);

    free (Layers);

    G_Tag++;

    for (j=0; j<BListSize (OutsMayFalse); j++)
    {
        Ind = BListElt (OutsMayFalse, j);
    }
}

```

```

NodeJ = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
GnlFreeNodesBddRefInfo (NodeJ);

NodeJ = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
GnlFreeNodesBddRefInfo (NodeJ);
}

FreeBddWorkSpace (Ws);

return (GNL_OK);
}
#endif

/* -----
/* GnlExtractPendingFunctions */ */
/* -----
/* Extracts the outputs which have different Bdd representatives. These */
/* outputs may lead to a functional difference. */ */
/* ----- */
void GnlExtractPendingFunctions (Gnl1, Gnl2, OutMayFalse)
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST      OutMayFalse;
{
    int           i;
    int           Ind;
    GNL_NODE Out1;
    GNL_NODE Out2;

    for (i=0; i<BListSize (OutMayFalse); i++)
    {
        Ind = BListElt (OutMayFalse, i);
        Out1 = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
        Out2 = (GNL_NODE)BListElt (G_OutputNodes2, Ind);

        /* If same Nodes then they are equivalent. */
        if (Out1 == Out2)
        {
            BListDelInsert (OutMayFalse, i+1);
            i--;
        }
        /*
        else
            fprintf (stderr, " EC: pending functions <%s> and <%s>\n",
                    GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)),
                    GnlVarName ((GNL_VAR)BListElt (G_Outputs2, Ind)));
        */
    }
}

/* -----
/* GnlCreateNewEcBddVarInputs */ */
/* ----- */
GNL_STATUS GnlCreateNewEcBddVarInputs (Node)
    GNL_NODE Node;
{

```

```

int i;
GNL_NODE SonI;
GNL_VAR Var;
char *NewStr;
BDD_VAR BddVar;
GNL_EC_CUT_POINT CutPoint;

if (GnlNodeTag (Node) == G_Tag)
    return (GNL_OK);
SetGnlNodeTag (Node, G_Tag);

if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return (GNL_OK);

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    Var = GnlNodeVar (Node);

    /* If the Var has no associated bdd var yet we create new one */
    if (GnlVarBdd (Var) == BDD_NULL)
    {
        if (GnlStrAppendIntCopy ("I", GLOB_BDD_WS->UniqueId+1, &NewStr))
            return (GNL_MEMORY_FULL);
        if (CreateBddVar (NewStr, &BddVar))
            return (GNL_MEMORY_FULL);
        SetGnlVarBdd (Var, UseBdd (BddVarBdd (BddVar)));
    }

    /* we create a new cut point and add it in the global list */
    /* of the cut points. */
    if (GnlCreateEcCutPoint (&CutPoint))
        return (GNL_MEMORY_FULL);
    SetGnlEcCutPointBddVar (CutPoint, BddVar);
    SetGnlEcCutPointNode (CutPoint, Node);
    if (BListAddElt (G_ListAllCutPoints, (int)CutPoint))
        return (GNL_MEMORY_FULL);
    }
    return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BlistElt (GnlNodeSons (Node), i);
    if (GnlCreateNewEcBddVarInputs (SonI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/* -----
/* CutPointsNodeIdentical
/* -----
int CutPointsNodeIdentical (CutPoint, Node)
    int CutPoint;
    int Node;
{

```

```

    return (GnlEcCutPointNode ((GNL_EC_CUT_POINT)CutPoint) == (GNL_NODE)Node) ;
}

/* -----
/* GnlCreateNewEcBddVar                               */
/* ----- */

GNL_STATUS GnlCreateNewEcBddVar (Node, OutsNum)
    GNL_NODE          Node;
    int              OutsNum;
{
    int                  i;
    GNL_NODE           SonI;
    char                *NewStr;
    BDD_VAR            BddVar;
    GNL_EC_CUT_POINT   CutPoint;
    int                 NodesRank;

    if (GnlNodeTag (Node) == G_Tag)
    {
#ifndef COMMON_CUTPOINTS_HEUR
        if (OutsNum == 2)
        {
            if (NodesRank = BListMemberOfList (G_ListOut1CutPoints, (int)Node,
                                              CutPointsNodeIdentical))
            {
                CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListOut1CutPoints,
NodesRank-1);
                if (BListAddElt (G_ListCommonCutPoints, (int)CutPoint))
                    return (GNL_MEMORY_FULL);
                BListDelShift (G_ListOut1CutPoints, NodesRank);
            }
        }
#endif
        return (GNL_OK);
    }

    SetGnlNodeTag (Node, G_Tag);

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlCreateNewEcBddVar (SonI, OutsNum))
            return (GNL_MEMORY_FULL);
    }

    if (GnlNodeBreakBddVar (Node) == (BDD_VAR)-1)
    {
        if (GnlStrAppendIntCopy ("B", GLOB_BDD_WS->UniqueId+1, &NewStr))
            return (GNL_MEMORY_FULL);
        if (CreateBddVar (NewStr, &BddVar))
            return (GNL_MEMORY_FULL);
        SetGnlNodeBreakBddVar (Node, BddVar);
    }
}

```

```

/* We create a new cut point and add it in the global list of the */
/* cut points. */                                                 */
if (GnlCreateEcCutPoint (&CutPoint))
    return (GNL_MEMORY_FULL);
SetGnlEcCutPointBddVar (CutPoint, BddVar);
SetGnlEcCutPointNode (CutPoint, Node);
if (BListAddElt (G_ListAllCutPoints, (int)CutPoint))
    return (GNL_MEMORY_FULL);
#ifndef COMMON_CUTPOINTS_HEUR
if (OutsNum == 1)
{
    if (BListAddElt (G_ListOut1CutPoints, (int)CutPoint))
        return (GNL_MEMORY_FULL);
}
else           /* OutsNum == 2 */                                */
{
    if (BListAddElt (G_ListOut2CutPoints, (int)CutPoint))
        return (GNL_MEMORY_FULL);
}
#endif
}

return (GNL_OK);
}

----- */
/* GnlCreateLocalBddNode */                                     */
/* ----- */                                                 */
GNL_STATUS GnlCreateLocalBddNode (RefGnl, Node, Top)
    GNL             RefGnl;
    GNL_NODE        Node;
    int             Top;
{
    BDD_VAR         BddVar;
    int             i;
    GNL_NODE        SonI;
    int             OverLimit;
    BDD             BddRes;
    GNL_NODE        InvNode;

    if (GnlNodeTag (Node) == G_Tag)
        return (GNL_OK);
    SetGnlNodeTag (Node, G_Tag);

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        if (GnlNodeSons (Node))
            SetGnlNodeBdd (Node, bdd_one ());
        else
            SetGnlNodeBdd (Node, bdd_zero ());

        if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
            return (GNL_MEMORY_FULL);

        return (GNL_OK);
    }
}

```

gnlec.c

```
if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    SetGnlNodeBdd (Node, UseBdd (GnlVarBdd (GnlNodeVar (Node))));

    if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

if ((BddVar = GnlNodeBreakBddVar (Node)) && Top)
{
    SetGnlNodeBdd (Node, UseBdd (BddVarBdd (BddVar)));

    if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BlistElt (GnlNodeSons (Node), i);
    if (GnlCreateLocalBddNode (RefGnl, SonI, 1))
        return (GNL_MEMORY_FULL);
}

if (ComputeBddOnNode (Node, LOCAL_MODE, &OverLimit, &BddRes))
    return (GNL_MEMORY_FULL);

SetGnlNodeBdd (Node, UseBdd (BddRes));

if (GnlNodeUpdateBddRef (RefGnl, REGULAR_MODE, Node, NULL))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/* -----
/* GnlBuildCutPointBdd
/* ----- */
GNL_STATUS GnlBuildCutPointBdd (RefGnl, CutPoint)
    GNL                    RefGnl;
    GNL_EC_CUT_POINT      CutPoint;
{
    int                     Index;
    GNL_NODE                CutPointNode;
    int                     NbNodes;

    if (!GnlEcCutPointBddFct (CutPoint))
    {
        CutPointNode = GnlEcCutPointNode (CutPoint);
        G_Tag++;
        if (GnlCreateLocalBddNode (RefGnl, CutPointNode, 0))
            return (GNL_MEMORY_FULL);
        SetGnlEcCutPointBddFct (CutPoint, UseBdd (GnlNodeBdd (CutPointNode)));
    }
}
```

gnlec.c

```
#ifdef LEAST_BDD_HEUR
    BddCountNodes (GnlEcCutPointBddFct (CutPoint), &NbNodes);
    SetGnlEcCutPointNbNode (CutPoint, NbNodes);
#endif

    return (GNL_OK);
}

/* -----
/* GnlBuildCutPointsBddFct
/* ----- */
GNL_STATUS GnlBuildCutPointsBddFct (RefGnl)
    GNL             RefGnl;
{
    int             i;
    GNL_EC_CUT_POINT CutPointI;

    /* Create Bdd for all the cut variables. */
    for (i=G_FirstBrkVarIndex; i<BListSize (G_ListAllCutPoints); i++)
    {
        CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
        if (GnlBuildCutPointBdd (RefGnl, CutPointI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/* -----
/* GnlResetCutPointsSupport
/* ----- */
void GnlResetCutPointsSupport (void)
{
    int             i;
    GNL_EC_CUT_POINT CutPointI;

    for (i=0; i<BListSize (G_ListAllCutPoints); i++)
    {
        CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
        SetGnlEcCutPointSupport (CutPointI, BDD_NULL);
        if (GnlNodeValue (GnlEcCutPointNode (CutPointI)) < 0)
            SetGnlNodeValue (GnlEcCutPointNode (CutPointI), 0);
    }
}

/* -----
/* SearchForIndexInBdd
/* ----- */
/*
int SearchForIndexInBdd (Bdd, Index)
    BDD             Bdd;
    int              Index;
{
    BDD_PTR          BddPtr;
```

gnlec.c

```
if (ConstantBdd (Bdd))
    return (0);

BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

if (IsBddTagCountBit (BddPtr))
    return (0);

BddTagCountBit (BddPtr);

if (GetBddPtrIndex (BddPtr) < Index)
    return (0);

if (GetBddPtrIndex (BddPtr) == Index)
    return (1);

return (SearchForIndexInBdd (GetBddPtrEdge1 (BddPtr), Index) ||
        SearchForIndexInBdd (GetBddPtrEdge0 (BddPtr), Index));
}

*/
/*-----*/
/* SearchForTagOccurBit */
/*-----*/
/*
int SearchForTagOccurBit (Bdd)
    BDD      Bdd;
{
    BDD_PTR  BddPtr;

    if (ConstantBdd (Bdd))
        return (0);

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

    if (IsBddTagCountBit (BddPtr))
        return (0);

    BddTagCountBit (BddPtr);

    if (IsBddTagOccurBit (BddPtr))
        return (1);

    return (SearchForTagOccurBit (GetBddPtrEdge1 (BddPtr)) ||
            SearchForTagOccurBit (GetBddPtrEdge0 (BddPtr)));
}

*/
/*-----*/
/* TotalResetTagOccurBit */
/*-----*/
/*
void TotalResetTagOccurBit (Bdd)
    BDD      Bdd;
{
    BDD_PTR  BddPtr;
```

```

if (ConstantBdd (Bdd))
    return;

BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

if (IsBddTagCountBit (BddPtr))
    return;

BddTagCountBit (BddPtr);

ResetBddTagOccurBit (BddPtr);

ResetTagOccurBit (GetBddPtrEdge1 (BddPtr));
ResetTagOccurBit (GetBddPtrEdge0 (BddPtr));
}

*/
/* -----
/* GnlGetBddSupportRec
/* -----
void GnlGetBddSupportRec (Bdd)
    BDD          Bdd;
{
    BDD_PTR        BddPtr;
    GNL_EC_CUT_POINT CutPoint;

    if (ConstantBdd (Bdd))
        return;

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

    /* Bdd with index less than 'G_FirstBrkVarIndex'. They correspond to */
    /* primary inputs.                                                 */
    if (GetBddPtrIndex (BddPtr) < G_FirstBrkVarIndex)
        return;

    if (IsBddTagCountBit (BddPtr))           /* Already traversed.      */
        return;

    BddTagCountBit (BddPtr);

    /* We mark the info associated to the bdd var.                  */
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints,
                                            GetBddPtrIndex (BddPtr));
    SetGnlEcCutPointSupport (CutPoint, bdd_one ());

    GnlGetBddSupportRec (GetBddPtrEdge1 (BddPtr));
    GnlGetBddSupportRec (GetBddPtrEdge0 (BddPtr));
}

*/
/* -----
/* GnlGetBddSupport
/* -----
void GnlGetBddSupport (Bdd)
    BDD          Bdd;
{

```

```

GnlResetCutPointsSupport ();
ResetTagCountBit (Bdd);
GnlGetBddSupportRec (Bdd);
ResetTagCountBit (Bdd);
}

#ifndef COMMON_CUTPOINTS_HEUR
/* ----- */
/* GnlExtractBestCutPoint0 */
/* ----- */
int GnlExtractBestCutPoint0 (RefGnl, Index)
    GNL RefGnl;
    int *Index;
{
    int i;
    int FindCutPoint1 = 1;
    int FindCutPoint2 = 1;
    int LastOut1Index;
    int LastOut2Index;
    GNL_EC_CUT_POINT CutPoint1;
    GNL_EC_CUT_POINT CutPoint2;
    GNL_EC_CUT_POINT CutPointI;

    LastOut1Index = BListSize (G_ListOut1CutPoints) - 1;
    LastOut2Index = BListSize (G_ListOut2CutPoints) - 1;

    while (1)
    {
        if (FindCutPoint1)
            CutPoint1 = NULL;
        if (FindCutPoint2)
            CutPoint2 = NULL;

        if (FindCutPoint1)
            for (i=LastOut1Index; i>=0; i--)
            {
                CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListOut1CutPoints, i);
                if (GnlEcCutPointSupport (CutPointI) &&
                    (GnlNodeValue (GnlEcCutPointNode (CutPointI)) >= 0))
                {
                    CutPoint1 = CutPointI;
                    FindCutPoint1 = 0;
                    LastOut1Index = i - 1;
                    break;
                }
            }

        if (FindCutPoint2)
            for (i=LastOut2Index; i>=0; i--)
            {
                CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListOut2CutPoints, i);
                if (GnlEcCutPointSupport (CutPointI) &&
                    (GnlNodeValue (GnlEcCutPointNode (CutPointI)) >= 0))
                {
                    CutPoint2 = CutPointI;
                    FindCutPoint2 = 0;
                    LastOut2Index = i - 1;
                }
            }
    }
}

```

```

        break;
    }
}

if (!CutPoint1 && !CutPoint2)
return (0);

for (i=BListSize (G_ListAllCutPoints)-1; i>=G_FirstBrkVarIndex; i--)
{
    CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
    if ((CutPointI == CutPoint1) || (CutPointI == CutPoint2))
    {
        if (GnlBuildCutPointBdd (RefGnl, CutPointI))
        {
#ifndef PRINT_CUTPOINTS_STAT
            fprintf (stderr, "Can't build CutPoint Bdd for Var = %s at phase
0\n",
                    BddVarName (GnlEcCutPointBddVar (CutPointI)));
#endif
        SetGnlNodeValue (GnlEcCutPointNode (CutPointI), -1);
        if (CutPointI == CutPoint1)
            FindCutPoint1 = 1;
        else
            FindCutPoint2 = 1;
        break;
    }
    *Index = i;
    return (1);
}
}
#endif
}

/* -----
/* GnlExtractBestCutPoint1
/* -----
int GnlExtractBestCutPoint1 (RefGnl, Index)
    GNL_RefGnl;
    int      *Index;
{
    int          i;
    GNL_EC_CUT_POINT      CutPoint;

    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, *Index);

    if (GnlNodeValue (GnlEcCutPointNode (CutPoint)) >= 0)
    {
#ifndef ALL_CUTPOINTS_TOGETHER
        if (GnlEcCutPointBddFct (CutPoint))
        return (1);

        if (!GnlBuildCutPointBdd (RefGnl, CutPoint))
        return (1);

        SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
#else

```

gnllec.c

```
    return (1);
#endif
}

for (i = *Index-1; i>=0; i--)
{
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
    if (!GnlEcCutPointSupport (CutPoint) ||
        (GnlNodeValue (GnlEcCutPointNode (CutPoint)) < 0))
        continue;

#ifndef ALL_CUTPOINTS_TOGETHER
    if (!GnlBuildCutPointBdd (RefGnl, CutPoint))
    {
        *Index = i;
        return (1);
    }
    SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
#else
    *Index = i;
    return (1);
#endif
#ifndef PRINT_CUTPOINTS_STAT
    fprintf (stderr, "Can't build CutPoint Bdd for Var = %s at phase 1\n",
            BddVarName (GnlEcCutPointBddVar (CutPoint)));
#endif
}
return (0);
}

/*
 * GnlExtractBestCutPoint2
 */
int GnlExtractBestCutPoint2 (RefGnl, Index)
    GNL_RefGnl;
    int      *Index;
{
    int          i;
    GNL_EC_CUT_POINT CutPointI;
    int          BestIndex = -1;
    int          MinNodes;

    /* BestIndex = G_FirstBrkVarIndex;           */
    MinNodes = 1000000;

    for (i=G_FirstBrkVarIndex; i<BListSize (G_ListAllCutPoints); i++)
    {
        CutPointI = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
#ifndef ALL_CUTPOINTS_TOGETHER
        if (GnlNodeValue (GnlEcCutPointNode (CutPointI)) < 0)
            continue;
#endif
        if (GnlEcCutPointSupport (CutPointI))
    {
```

```

#ifndef ALL_CUTPOINTS_TOGETHER
    if (GnlBuildCutPointBdd (RefGnl, CutPointI))
    {
        SetGnlNodeValue (GnlEcCutPointNode (CutPointI), -1);
#endif PRINT_CUTPOINTS_STAT
        fprintf (stderr, "Can't build CutPoint Bdd for Var = %s at phase
2\n",
                BddVarName (GnlEcCutPointBddVar (CutPointI)));
#endif
        continue;
    }
#endif

    if (GnlEcCutPointNbNode (CutPointI) <= MinNodes)
    {
        MinNodes = GnlEcCutPointNbNode (CutPointI);
        BestIndex = i;
    }
}

if (BestIndex >= 0)
{
    *Index = BestIndex;
    return (1);
}
else
    return (0);
}

/* -----
/* PrintCutPointsMap
/* -----
void PrintCutPointsMap (void)
{
    int
        j;
    GNL_EC_CUT_POINT      CutPoint;

    fprintf (stderr, "Input Break Vars:\n");
    for (j=0; j<G_FirstBrkVarIndex; j++)
    {
        CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, j);
        if (GnlEcCutPointSupport (CutPoint))
            fprintf (stderr, "      Var = %s, NbNodes = %d\n",
                    BddVarName (GnlEcCutPointBddVar (CutPoint)),
                    GnlEcCutPointNbNode (CutPoint));
    }

#endif COMMON_CUTPOINTS_HEUR
    fprintf (stderr, "Out1 Break Vars:\n");
    for (j=0; j<BListSize (G_ListOut1CutPoints); j++)
    {
        CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListOut1CutPoints, j);
        if (GnlEcCutPointSupport (CutPoint))
            fprintf (stderr, "      Var = %s, NbNodes = %d\n",
                    BddVarName (GnlEcCutPointBddVar (CutPoint)),
                    GnlEcCutPointNbNode (CutPoint));
    }
}

```

```

}

fprintf (stderr, "Out2 Break Vars:\n");
for (j=0; j<BListSize (G_ListOut2CutPoints); j++)
{
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListOut2CutPoints, j);
    if (GnlEcCutPointSupport (CutPoint))
        fprintf (stderr, "      Var = %s, NbNodes = %d\n",
                 BddVarName (GnlEcCutPointBddVar (CutPoint)),
                 GnlEcCutPointNbNode (CutPoint));
}

fprintf (stderr, "Common Break Vars:\n");
for (j=0; j<BListSize (G_ListCommonCutPoints); j++)
{
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListCommonCutPoints, j);
    if (GnlEcCutPointSupport (CutPoint))
        fprintf (stderr, "      Var = %s, NbNodes = %d\n",
                 BddVarName (GnlEcCutPointBddVar (CutPoint)),
                 GnlEcCutPointNbNode (CutPoint));
}
#else
fprintf (stderr, "Other Break Vars:\n");
for (j=G_FirstBrkVarIndex; j<BListSize (G_ListAllCutPoints); j++)
{
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, j);
    if (GnlEcCutPointSupport (CutPoint))
        fprintf (stderr, "      Var = %s, NbNodes = %d\n",
                 BddVarName (GnlEcCutPointBddVar (CutPoint)),
                 GnlEcCutPointNbNode (CutPoint));
}
#endif
}

/*-----*/
/* ComputeConstraints */
/*-----*/
int ComputeConstraints (Bdd)
    BDD          Bdd;
{
    BDD_PTR      BddPtr;
    GNL_NODE     Node;
    BLIST        NewListConstraints;
    int          Index;
    int          ToContinue;

    if (Bdd == bdd_one ())
    {
        if (BListAddElt (G_ListBadPatterns, (int)G_ListConstraints))
            return (0);

        if (BListCopyNoEltCr (G_ListConstraints, &NewListConstraints))
            return (0);

        G_ListConstraints = NewListConstraints;
    }

    return (1);
}

```

```

}

if (ConstantBdd (Bdd))
    return (1);

BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

if (IsBddTagOccurBit (BddPtr))
    return (1);

BddTagOccurBit (BddPtr);

Index = GetBddPtrIndex (BddPtr);
Node = GnlEcCutPointNode ((GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints,
Index));

if (IsBddTypeINV (Bdd))
{
    if (BListAddElt (G_ListConstraints, (int)GnlNodePosConstraint (Node)))
        return (0);
    ToContinue = ComputeConstraints (ComplementBDD (GetBddPtrEdge1
(BddPtr)));
    BListDelInsert (G_ListConstraints, BListSize (G_ListConstraints));
    if (!ToContinue)
        return (0);

    if (BListAddElt (G_ListConstraints, (int)GnlNodeNegConstraint (Node)))
        return (0);
    ToContinue = ComputeConstraints (ComplementBDD (GetBddPtrEdge0
(BddPtr)));
    BListDelInsert (G_ListConstraints, BListSize (G_ListConstraints));
    return (ToContinue);
}
else
{
    if (BListAddElt (G_ListConstraints, (int)GnlNodePosConstraint (Node)))
        return (0);
    ToContinue = ComputeConstraints (GetBddPtrEdge1 (BddPtr));
    BListDelInsert (G_ListConstraints, BListSize (G_ListConstraints));
    if (!ToContinue)
        return (0);

    if (BListAddElt (G_ListConstraints, (int)GnlNodeNegConstraint (Node)))
        return (0);
    ToContinue = ComputeConstraints (GetBddPtrEdge0 (BddPtr));
    BListDelInsert (G_ListConstraints, BListSize (G_ListConstraints));
    return (ToContinue);
}
}

#endif XOR_ANALYSIS
/* -----
/* GnlGetTransitiveBddSupportRec
/* -----
/* Computes the transitive support of 'Bdd' by taking into account in
/* the support the intermediate cut variables. */
/* -----

```

gnlec.c

```
void GnlGetTransitiveBddSupportRec (Bdd)
    BDD          Bdd;
{
    BDD_PTR      BddPtr;
    GNL_EC_CUT_POINT CutPoint;
    BDD          BddCutPoint;

    if (ConstantBdd (Bdd))
        return;

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);

    if (IsBddTagCountBit (BddPtr))           /* Already traversed. */
        return;

    BddTagCountBit (BddPtr);

    /* We mark the info associated to the bdd var. */
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints,
                                            GetBddPtrIndex (BddPtr));

    if (GnlEcCutPointSupport (CutPoint) != bdd_one ())
    {
        /* Bdd cut point not already traversed so we do it. */
        SetGnlEcCutPointSupport (CutPoint, bdd_one ());
        BddCutPoint = GnlEcCutPointBddFct (CutPoint);
        if (BddCutPoint)
            GnlGetTransitiveBddSupportRec (BddCutPoint);
    }

    GnlGetTransitiveBddSupportRec (GetBddPtrEdge1 (BddPtr));
    GnlGetTransitiveBddSupportRec (GetBddPtrEdge0 (BddPtr));
}

/*
 * -----
 * GnlGetTransitiveBddSupport
 * -----
 * Computes the transitive support of 'Bdd' by taking into account in
 * the support the intermediate cut variables.
 * The support is returned thru the list 'ListSupport'.
 */
GNL_STATUS GnlGetTransitiveBddSupport (Bdd, ListSupport)
    BDD          Bdd;
    BLIST      *ListSupport;
{
    int          i;
    GNL_EC_CUT_POINT CutPoint;

    GnlResetCutPointsSupport ();
    ResetTagCountBit (Bdd);
    GnlGetTransitiveBddSupportRec (Bdd);
    ResetTagCountBit (Bdd);

    if (BListCreateWithSize (1, ListSupport))
```

gnlec.c

```
    return (GNL_MEMORY_FULL);

for (i=BListSize (G_ListAllCutPoints)-1; i>=0; i--)
{
    CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, i);
    if (GnlEcCutPointSupport (CutPoint) == bdd_one ())
    {
        if (BListAddElt (*ListSupport, (int)CutPoint))
            return (GNL_MEMORY_FULL);
    }
}

GnlResetCutPointsSupport ();

return (GNL_OK);
}

/* -----
/* GnlPrintOnePath
/* -----
void GnlPrintOnePath (ListSupport)
    BLIST      ListSupport;
{
    int                  i;
    GNL_EC_CUT_POINT    CutPointI;
    int                  Size;
    char                 ValueChar;

    GNL_VAR              NodesVar;
    char                 *Name;

    Size = 0;
    for (i=0; i<BListSize (ListSupport); i++)
    {
        CutPointI = (GNL_EC_CUT_POINT)BListElt (ListSupport, i);
        if (!GnlEcCutPointSupport (CutPointI))
            continue;
        ValueChar = (GnlEcCutPointSupport (CutPointI) == bdd_one () ? '1'
                     : '0');
        if (!GnlEcCutPointSupport (CutPointI))
            ValueChar = '-';

        if (!(NodesVar = GnlNodeVar (GnlEcCutPointNode (CutPointI))))
            Name = "No var";
        else
            Name = GnlVarName (NodesVar);

        fprintf (stderr, "[%d %s %c]", GnlEcCutPointBddVar (CutPointI)->Id,
                 Name, ValueChar);
        Size++;
    }

    fprintf (stderr, "\nSize = %d\n", Size);
}
```

```

/* ----- */
/* GnlRecomposeCutPoints */
/* ----- */

GNL_STATUS GnlRecomposeCutPoints (CutPoint, BddValue, Conflict)
    GNL_EC_CUT_POINT      CutPoint;
    BDD                  BddValue;
    int                  *Conflict;
{
    int                  Index;
    int                  j;
    BDD                  BddJ;
    BDD_STATUS           Status;
    int                  IsConflict = 0;
    GNL_EC_CUT_POINT     CutPointJ;
    BDD                  BddRes;

Index = GnlEcCutPointBddVar (CutPoint) ->Id;

for (j=0; j<BListSize (G_TransitiveXorSupport); j++)
{
    CutPointJ = (GNL_EC_CUT_POINT)
        BListElt (G_TransitiveXorSupport, j);

    if (CutPointJ == CutPoint)
        return (GNL_OK);

    BddJ = (BDD)BListElt (G_ListConflictBdd, j);

    if (BddJ == BDD_NULL)
        continue;

    if (Status = bdd_compose (BddJ, Index, BddValue, &BddRes))
        return (GNL_MEMORY_FULL);

    if (BddJ == BddRes)
        continue;

    BddJ = BListElt (G_ListConflictBdd, j) = (int)BddRes;

    /* If there is a conflict value for cut point J then we */
    /* stop. */
    if (ConstantBdd (BddJ))
    {
        /* If there is a value for 'CutPointJ' different of its */
        /* composition result then there is a conflict. */
        if (GnlEcCutPointSupport (CutPointJ) &&
            (BddJ != GnlEcCutPointSupport (CutPointJ)))
        {
            *Conflict = 1;
            return (GNL_OK);
        }
        if (!GnlEcCutPointSupport (CutPointJ))
        {
            if (GnlRecomposeCutPoints (CutPointJ, BddJ,
                &IsConflict))

```

```

        return (GNL_MEMORY_FULL);
    if (IsConflict)
    {
        *Conflict = 1;
        return (GNL_OK);
    }
}
}

*Conflict = 0;

return (GNL_OK);
}

/*
/* GnlCheckConflict
*/
/*
GNL_STATUS GnlCheckConflict (Conflict, FalsePath)
    int          *Conflict;
    int          *FalsePath;
{
    int          i;
    int          j;
    GNL_EC_CUT_POINT CutPointI;
    GNL_EC_CUT_POINT CutPointJ;
    int          IndexI;
    BDD          BddJ;
    BDD          BddRes;
    BDD_STATUS   Status;
    int          Path;
    BDD          BddExpr;
    BDD          BddProd;
    int          Index;

*FalsePath = 1000000;

/* We add the forest of Bdds in the list 'G_TransitiveXorSupport'      */
BSize (G_ListConflictBdd) = 0;
for (i=0; i<BListSize (G_TransitiveXorSupport); i++)
{
    CutPointI = (GNL_EC_CUT_POINT)
                BListElt (G_TransitiveXorSupport, i);
    if (BListAddElt (G_ListConflictBdd,
                     (int)GnlEcCutPointBddFct (CutPointI)))
        return (GNL_MEMORY_FULL);
}

/*
GnlPrintOnePath (G_TransitiveXorSupport);
*/
/*
/* we substitute in a top-down approach each sub bdds in order to      */
/* expect a conflict.                                                 */
Path = -1;

```

```

for (i=0; i<BListSize (G_TransitiveXorSupport); i++)
{
    CutPointI = (GNL_EC_CUT_POINT)BListElt (G_TransitiveXorSupport, i);

    /* If it is not a Var on the current Xor path. */
    if (!GnlEcCutPointSupport (CutPointI))
        continue;

    Path++;

    IndexI = GnlEcCutPointBddVar (CutPointI)->Id;

    for (j=0; j<i; j++)
    {
        CutPointJ = (GNL_EC_CUT_POINT)
                    BListElt (G_TransitiveXorSupport, j);
        BddJ = (BDD)BListElt (G_ListConflictBdd, j);

        if (BddJ == BDD_NULL)
            continue;

        if (Status = bdd_compose (BddJ, IndexI,
                                  GnlEcCutPointSupport (CutPointI),
                                  &BddRes))
            return (GNL_MEMORY_FULL);

        if (BddJ == BddRes)
            continue;

        BddJ = BListElt (G_ListConflictBdd, j) = (int)BddRes;

        /* If there is a conflict value for cut point J then we */
        /* stop. */
        if (ConstantBdd (BddJ))
        {
            /* If there is a value for 'CutPointJ' different of its*/
            /* composition result then there is a conflict. */
            if (GnlEcCutPointSupport (CutPointJ) &&
                (BddJ != GnlEcCutPointSupport (CutPointJ)))
            {
                *Conflict = 1;
                *FalsePath = Path;
                return (GNL_OK);
            }
        }

        if (!GnlEcCutPointSupport (CutPointJ))
        {
            if (GnlRecomposeCutPoints (CutPointJ, BddJ,
                                      Conflict))
                return (GNL_MEMORY_FULL);
            if (*Conflict == 1)
            {
                *FalsePath = Path;
                return (GNL_OK);
            }
        }
    }
}

```

```

        }
    }

*Conflict = 0;
for (j=0; j<BListSize (G_ListConflictBdd); j++)
{
    BddJ = (BDD)BListElt (G_ListConflictBdd, j);

    if (BddJ == BDD_NULL)
        continue;

    /* There is an expression which is not constant so we cannot      */
    /* that there is no uncompatibility.                                */
    if (!ConstantBdd (BddJ))
    {
        *Conflict = -1;
        break;
    }
}

if (*Conflict == 0)
    return (GNL_OK);

BddProd = bdd_one ();

for (j=0; j<BListSize (G_TransitiveXorSupport); j++)
{
    CutPointJ = (GNL_EC_CUT_POINT)
                BListElt (G_TransitiveXorSupport, j);

    /* If it is not a Var on the current Xor path.                  */
    if (!GnlEcCutPointSupport (CutPointJ))
        continue;

    BddJ = (BDD)BListElt (G_ListConflictBdd, j);

    if (BddJ == BDD_NULL)
        continue;

    if (!ConstantBdd (BddJ))
    {
        if (GnlEcCutPointSupport (CutPointJ) == bdd_one ())
            BddExpr = BddJ;
        else
            BddExpr = ComplementBDD (BddJ);

        if ((Status = bdd_and (BddProd, BddExpr, &BddRes)))
            return (GNL_MEMORY_FULL);

        BddProd = BddRes;

        /* If the product is (0) so there is a conflict      */
        if (BddProd == bdd_zero ())
        {
            *Conflict = 1;
        }
    }
}

```

```

        return (GNL_OK);
    }
}

if (BddProd == bdd_one ())
{
    *Conflict = 0;
    return (GNL_OK);
}

while (1)
{
    Index = GetBddPtrIndex ((BDD_PTR)GetBddPtrFromBdd (BddProd));
    if (Index < G_FirstBrkVarIndex)
    {
        *Conflict = 0;
        return (GNL_OK);
    }

    for (j=0; j<BListSize (G_TransitiveXorSupport); j++)
    {
        CutPointJ = (GNL_EC_CUT_POINT)
            BListElt (G_TransitiveXorSupport, j);

        if (GnlEcCutPointBddVar (CutPointJ)->Id == Index)
            break;
    }
    BddJ = (BDD)BListElt (G_ListConflictBdd, j);

    if (Status = bdd_compose (BddProd, Index, BddJ, &BddRes))
        return (GNL_MEMORY_FULL);

    BddProd = BddRes;

    if (BddProd == bdd_zero ())
    {
        *Conflict = 1;
        return (GNL_OK);
    }

    if (BddProd == bdd_one ())
    {
        *Conflict = 0;
        return (GNL_OK);
    }
}

return (GNL_OK);
}

/*
 * -----
 * GnlExtractBddOnePath
 * -----
 */
GNL_STATUS GnlExtractBddOnePath (Bdd, Path, Correct)
    BDD          Bdd;

```

```

int          Path;
int          *Correct;
{
    BDD_PTR           BddPtr;
    int               Index;
    GNL_EC_CUT_POINT CutPoint;
    int               Conflict = 0;
    int               FalsePath;

    if (*Correct < 1)
        return (GNL_OK);

    if (Path > G_FalsePath)
        return (GNL_OK);
    G_FalsePath = 1000000;

    if (Bdd == bdd_one ())
    {
        if (G_CheckConflict >= CHECK_MAX_CONFLICTS)
        {
            *Correct = -1;
            return (GNL_OK);
        }

        G_CheckConflict++;

        if (GnlCheckConflict (&Conflict, &G_FalsePath))
            return (GNL_MEMORY_FULL);

        if (Conflict == 0)
        {
            fprintf (stderr, "error !\n");
            *Correct = 0;
            return (GNL_OK);
        }

        if (Conflict == -1)
        {
            *Correct = -1;
            return (GNL_OK);
        }
    }

    *Correct = 1;
    return (GNL_OK);
}

if (ConstantBdd (Bdd))
{
    return (GNL_OK);
}

BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);
Index = GetBddPtrIndex (BddPtr);
CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, Index);

if (IsBddTypeINV (Bdd))

```

```

{
    SetGnlEcCutPointSupport (CutPoint, bdd_zero ());
    if (GnlExtractBddOnePath (ComplementBDD (GetBddPtrEdge0 (BddPtr)),
                              Path+1, Correct))
        return (GNL_MEMORY_FULL);
}
else
{
    SetGnlEcCutPointSupport (CutPoint, bdd_zero ());
    if (GnlExtractBddOnePath (GetBddPtrEdge0 (BddPtr),
                              Path+1, Correct))
        return (GNL_MEMORY_FULL);
}

/* Case of error or undetermination. */
if (*Correct < 1)
    return (GNL_OK);

if (IsBddTypeINV (Bdd))
{
    SetGnlEcCutPointSupport (CutPoint, bdd_one ());
    if (GnlExtractBddOnePath (ComplementBDD (GetBddPtrEdge1 (BddPtr)),
                              Path+1, Correct))
        return (GNL_MEMORY_FULL);
}
else
{
    SetGnlEcCutPointSupport (CutPoint, bdd_one ());
    if (GnlExtractBddOnePath (GetBddPtrEdge1 (BddPtr),
                              Path+1, Correct))
        return (GNL_MEMORY_FULL);
}

SetGnlEcCutPointSupport (CutPoint, BDD_NULL);

return (GNL_OK);
}

/*
 * -----
 * GnlAnalyzeBddXor
 * -----
 * GNL_STATUS GnlAnalyzeBddXor (BddXor, Correct)
 * BDD          BddXor;
 * int          *Correct;
 {
     BLIST      XorSupport;

     if (GnlGetTransitiveBddSupport (BddXor, &XorSupport))
         return (GNL_MEMORY_FULL);

     G_TransitiveXorSupport = XorSupport;
     if (BListCreateWithSize (BListSize (G_TransitiveXorSupport),
                            &G_ListConflictBdd))
         return (GNL_MEMORY_FULL);
}

```

```

G_CheckConflict = 0;
*Correct = 1;           /* we suppose it is correct          */
if (GnlExtractBddOnePath (BddXor, 0, Correct))           */
    return (GNL_MEMORY_FULL);                                */

fprintf (stderr, "Check Conflicts = %d\n", G_CheckConflict);

return (GNL_OK);
}
#endif

#ifndef IMPLICATION_CHECK
/* ----- */                                     */
/* GnlResolveTwoNodes                         */
/* ----- */                                     */
GNL_STATUS GnlResolveTwoNodes (NbMaxUsedNodes, RefGnl, Node1, Node2, Invert)
    int                      NbMaxUsedNodes;
    GNL                      RefGnl;
    GNL_NODE                Node1;
    GNL_NODE                Node2;
    int                      *Invert;
{
    int                      i;
    BDD_WS                  Ws;
    BDD                      Bdd1;
    BDD                      Bdd2;
    BDD                      BddXor;
    BDD_PTR                 BddXorPtr;
    BDD                      BddRes;
    BDD                      VarBddFct;
    int                      Index;
    int                      newIndex;
    GNL_EC_CUT_POINT        CutPoint;
    GNL_NODE                CutPointNode;
    int                      SameBddXor;
    int                      Phase = 0;
    int                      Correct;

G_BadPatternsFound = 0;

if ((GnlNodeOp (Node1) == GNL_VARIABLE) &&
    (GnlNodeOp (Node2) == GNL_VARIABLE))
    return (GNL_ERROR_DETECTED);

while (1)
{
    if (BListCreateWithSize (1, &G_ListAllCutPoints))
        return (GNL_MEMORY_FULL);
#endif COMMON_CUTPOINTS_HEUR
    if (BListCreateWithSize (1, &G_ListOut1CutPoints))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &G_ListOut2CutPoints))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &G_ListCommonCutPoints))
        return (GNL_MEMORY_FULL);
}

```

gnlec.c

```
#endif

if (InitBddWorkSpaceInvOrder (100, &Ws))
    return (GNL_MEMORY_FULL);

/* we authorize a maximum number of used nodes. */
SetNbMaxUsedNode (NbMaxUsedNodes);

/* We create bdd var for primary inputs. */
G_Tag++;
if (GnlCreateNewEcBddVarInputs (Node1))
    return (GNL_MEMORY_FULL);
if (GnlCreateNewEcBddVarInputs (Node2))
    return (GNL_MEMORY_FULL);

/*
for (i=0; i<BListSize (GnlRelevantInputs (RefGnl)); i++)
    if (GnlCreateNewEcBddVarInputs ((GNL_NODE) GnlVarDads
((GNL_VAR) BListElt (GnlRelevantInputs (RefGnl), i))))
        return (GNL_MEMORY_FULL);
*/
G_FirstBrkVarIndex = Ws->UniqueId + 1;

/* We create bdd var for local variables. */
G_Tag++;
if (GnlCreateNewEcBddVar (Node1, 1))
    return (GNL_MEMORY_FULL);
if (GnlCreateNewEcBddVar (Node2, 2))
    return (GNL_MEMORY_FULL);

G_Tag++;
if (GnlCreateLocalBddNode (RefGnl, Node1, 0))
    return (GNL_MEMORY_FULL);
if (GnlCreateLocalBddNode (RefGnl, Node2, 0))
    return (GNL_MEMORY_FULL);

Bdd1 = GnlNodeBdd (Node1);
Bdd2 = GnlNodeBdd (Node2);

if (bdd_xor (Bdd1, Bdd2, &BddXor))
{
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_UNRESOLVED_AT_XOR_BUILD);
}

if (BddXor == bdd_zero ())
{
    if (ReconnectNode (Node2, Node1, Invert))
        return (GNL_MEMORY_FULL);

    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_EQUIVALENT_AT_XOR_BUILD);
}

if (BddXor == bdd_one ())
```

```

{
if (ReconnectInvNode (RefGnl, Node2, Node1, Invert))
    return (GNL_MEMORY_FULL);

G_BadPatternsFound = -1;

GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
return (GNL_ERROR_DETECTED_AT_XOR_BUILD);
}

#ifndef ALL_CUTPOINTS_TOGETHER
if (GnlBuildCutPointsBddFct (RefGnl))
{
GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
return (GNL_UNRESOLVED_AT_BUILD_CUTPOINT);
}
#endif

SameBddXor = 0;

while (1)
{
if (!SameBddXor)
{
BddXorPtr = GetBddPtr (BddXor);
Index = GetBddPtrIndex (BddXorPtr);
if (Index < G_FirstBrkVarIndex)
{
if (!G_ListBadPatterns &&
    BListCreateWithSize (1000, &G_ListBadPatterns))
return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);
if (BListCreateWithSize (G_FirstBrkVarIndex, &G_ListConstraints))
return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);
G_BadPatternsFound = 1;
ResetTagOccurBit (BddXor);
ComputeConstraints (BddXor);
ResetTagOccurBit (BddXor);
BListQuickDelete (&G_ListConstraints);

GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);
}
}

/* We use the first heuristic. */
if (Phase == 0)
{
if (!SameBddXor)
{
GnlGetBddSupport (BddXor);
#endif PRINT_CUTPOINTS_STAT
PrintCutPointsMap ();
#endif
}
#endif COMMON_CUTPOINTS_HEUR
if (!GnlExtractBestCutPoint0 (RefGnl, &NewIndex))

```

```

        Phase = 1;
    else
        Index = NewIndex;
    }

/* We use the second heuristic. */
if (Phase == 1)
{
    if (!SameBddXor)
    {
        GnlGetBddSupport (BddXor);
#ifndef PRINT_CUTPOINTS_STAT
        PrintCutPointsMap ();
#endif
    }
}
#endif
if (!GnlExtractBestCutPoint1 (RefGnl, &Index))
{
#endif LEAST_BDD_HEUR
    Phase = 2;
    break;
#else
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_UNRESOLVED_AT_EXTRACTION);
#endif
}

#endif LEAST_BDD_HEUR
/* We use the third heuristic. */
if (Phase == 2)
{
    if (!SameBddXor)
    {
        GnlGetBddSupport (BddXor);
#ifndef PRINT_CUTPOINTS_STAT
        PrintCutPointsMap ();
#endif
    }
}
if (!GnlExtractBestCutPoint2 (RefGnl, &Index))
{
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_UNRESOLVED_AT_EXTRACTION);
}
}

CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, Index);
VarBddFct = GnlEcCutPointBddFct (CutPoint);

if (bdd_compose (BddXor, Index, VarBddFct, &BddRes))
{
#endif PRINT_COMPOSE_STAT
    fprintf (stderr, "Tried to compose Var = %s (NbNodes = %d) at phase
%d\n",
            BddVarName (GnlEcCutPointBddVar (CutPoint)),
            GnlEcCutPointNbNode (CutPoint), Phase);
}

```

```

#endif
#ifndef LEAST_BDD_HEUR
    if (Phase == 0)
    {
#endif
#ifndef COMPOSE_UNTIL_FIRST_FAILURE
        SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
        SameBddXor = 1;
        continue;
#else
        /* We try the second heuristic. */
        Phase = 2;
#endif
#endif
}

#ifndef COMMON_CUTPOINTS_HEUR
    if (Phase == 1)
    {
        Phase = 2;
        break;
    }
#endif
else
{
#endif
#ifndef XOR_ANALYSIS
    SetNbMaxUsedNode (GetNbMaxUsedNode ()+200000);

    if (GnlAnalyzeBddXor (BddXor, &Correct))
        return (GNL_MEMORY_FULL);

    if (Correct == 1)
    {
        return (GNL_EQUIVALENT_PHASE0 + Phase);
    }
    if (Correct == 0)
    {
        return (GNL_ERROR_DETECTED_PHASE0 + Phase);
    }
    if (Correct == -1)
    {
        return (GNL_UNRESOLVED);
    }
#else
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_UNRESOLVED);
#endif
}
break;
#endif
#ifndef COMPOSE_UNTIL_FIRST_FAILURE
    SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
    SameBddXor = 1;
    continue;
#else
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_UNRESOLVED);
#endif
#endif

```

```

        }

BddXor = BddRes;
SameBddXor = 0;

#ifndef PRINT_COMPOSE_STAT
    fprintf (stderr, "Composed Var = %s (NbNodes = %d) at phase %d\n",
             BddVarName (GnlEcCutPointBddVar (CutPoint)),
             GnlEcCutPointNbNode (CutPoint), Phase);
#endif

if (BddXor == bdd_zero ())
{
    if (ReconnectNode (Node2, Node1, Invert))
        return (GNL_MEMORY_FULL);

    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
    return (GNL_EQUIVALENT_PHASE0 + Phase);
}
else
{
    if (BddXor == bdd_one ())
    {
        if (ReconnectInvNode (RefGnl, Node2, Node1, Invert))
            return (GNL_MEMORY_FULL);

        G_BadPatternsFound = -1;

        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
        return (GNL_ERROR_DETECTED_PHASE0 + Phase);
    }
}

GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

BListDelete (&G_ListAllCutPoints, GnlCutPointFree);
#ifndef COMMON_CUTPOINTS_HEUR
    BListQuickDelete (&G_ListOut1CutPoints);
    BListQuickDelete (&G_ListOut2CutPoints);
    BListQuickDelete (&G_ListCommonCutPoints);
#endif
}
#else
/* -----
/* GnlResolveTwoNodes
/* -----
GNL_STATUS GnlResolveTwoNodes (NbMaxUsedNodes, RefGnl, FirstNode, SecondNode,
Invert)
{
    int                      NbMaxUsedNodes;
    GNL                      RefGnl;
    GNL_NODE                FirstNode;
    GNL_NODE                SecondNode;
    int                      *Invert;
{
    int                      i;
    BDD_WS                  Ws;
    BDD                     Bdd1;
    BDD                     Bdd2;

```

```

BDD           BddImpl;
BDD_PTR      BddImplPtr;
BDD          BddRes;
BDD          VarBddFct;
int          Index;
int          NewIndex;
GNL_EC_CUT_POINT   CutPoint;
GNL_NODE     CutPointNode;
int          SameBddImpl;
int          Phase = 0;
int          Correct;
int          Direction = 1;
int          LeftImplRight = -1;
int          RightImplLeft = -1;
int          Node1 = FirstNode;
GNL_NODE    Node2 = SecondNode;

/*
int          IndexExists;
int          TagOccurBitExists;
*/,

G_BadPatternsFound = 0;

if ((GnlNodeOp (Node1) == GNL_VARIABLE) &&
    (GnlNodeOp (Node2) == GNL_VARIABLE))
    return (GNL_ERROR_DETECTED);

while (Direction < 3)
{
    if (BListCreateWithSize (1, &G_ListAllCutPoints))
        return (GNL_MEMORY_FULL);
#ifndef COMMON_CUTPOINTS_HEUR
    if (BListCreateWithSize (1, &G_ListOut1CutPoints))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &G_ListOut2CutPoints))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &G_ListCommonCutPoints))
        return (GNL_MEMORY_FULL);
#endif
    if (Direction == 2)
    {
        Node1 = SecondNode;
        Node2 = FirstNode;
    }

    if (InitBddWorkSpaceInvOrder (100, &Ws))
        return (GNL_MEMORY_FULL);

    /* we authorize a maximum number of used nodes. */ *
    SetNbMaxUsedNode (NbMaxUsedNodes);

    /* We create bdd var for primary inputs. */ *
    G_Tag++;
    if (GnlCreateNewEcBddVarInputs (Node1))
        return (GNL_MEMORY_FULL);

```

```

    if (GnlCreateNewEcBddVarInputs (Node2))
        return (GNL_MEMORY_FULL);

    /*
    for (i=0; i<BListSize (GnlRelevantInputs (RefGnl)); i++)
        if (GnlCreateNewEcBddVarInputs ((GNL_NODE) GnlVarDads
((GNL_VAR) BListElt (GnlRelevantInputs (RefGnl), i))))
            return (GNL_MEMORY_FULL);
    */

    G_FirstBrkVarIndex = Ws->UniqueId + 1;

    /* We create bdd var for local variables. */
    G_Tag++;
    if (GnlCreateNewEcBddVar (Node1, 1))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNewEcBddVar (Node2, 2))
        return (GNL_MEMORY_FULL);

    G_Tag++;
    if (GnlCreateLocalBddNode (RefGnl, Node1, 0))
        return (GNL_MEMORY_FULL);
    if (GnlCreateLocalBddNode (RefGnl, Node2, 0))
        return (GNL_MEMORY_FULL);

    Bdd1 = GnlNodeBdd (Node1);
    Bdd2 = GnlNodeBdd (Node2);

    if (bdd_impl (Bdd1, Bdd2, &BddImpl))
    {
        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
        return (GNL_UNRESOLVED_AT_XOR_BUILD);
    }

    if (BddImpl == bdd_one ())
    {
        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

        if (Direction == 1)
            LeftImplRight = 1;
        else
            RightImplLeft = 1;

        Direction++;

        continue;
    }

#endif ALL_CUTPOINTS_TOGETHER
    if (GnlBuildCutPointsBddFct (RefGnl))
    {
        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);
        return (GNL_UNRESOLVED_AT_BUILD_CUTPOINT);
    }
#endif

```

```

SameBddImpl = 0;

while (1)
{
    if (!SameBddImpl)
    {
        BddImplPtr = GetBddPtr (BddImpl);
        Index = GetBddPtrIndex (BddImplPtr);
        if (Index < G_FirstBrkVarIndex)
        {
            if (!G_ListBadPatterns &&
                BListCreateWithSize (1000, &G_ListBadPatterns))
                return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);
            if (BListCreateWithSize (G_FirstBrkVarIndex, &G_ListConstraints))
                return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);
            G_BadPatternsFound = 1;
            ResetTagOccurBit (BddImpl);
            ComputeConstraints (ComplementBDD (BddImpl));
            ResetTagOccurBit (BddImpl);
            BListQuickDelete (&G_ListConstraints);

            GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

            if (Direction == 1)
                LeftImplRight = 0;
            else
                RightImplLeft = 0;

            Direction = 3;

            break;
        }
    }

/* We use the first heuristic. */
if (Phase == 0)
{
    if (!SameBddImpl)
    {
        GnlGetBddSupport (BddImpl);
#ifndef PRINT_CUTPOINTS_STAT
        PrintCutPointsMap ();
#endif
    }
}

#ifndef COMMON_CUTPOINTS_HEUR
    if (!GnlExtractBestCutPoint0 (RefGnl, &NewIndex))
        Phase = 1;
    else
        Index = NewIndex;
}

/* We use the second heuristic. */
if (Phase == 1)
{
    if (!SameBddImpl)
    {
        GnlGetBddSupport (BddImpl);
}

```

```

#ifndef PRINT_CUTPOINTS_STAT
    PrintCutPointsMap ();
#endif
}
#endif
    if (!GnlExtractBestCutPoint1 (RefGnl, &Index))
    {
#endif LEAST_BDD_HEUR
    Phase = 2;
    break;
#else
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

    if (Direction == 1)
    LeftImplRight = -1;
    else
    RightImplLeft = -1;

    Direction++;

    break;
#endif
}
}

#endif LEAST_BDD_HEUR
/* We use the third heuristic. */
if (Phase == 2)
{
    if (!SameBddImpl)
    {
        GnlGetBddSupport (BddImpl);
#endif PRINT_CUTPOINTS_STAT
        PrintCutPointsMap ();
#endif
    }
    if (!GnlExtractBestCutPoint2 (RefGnl, &Index))
    {
        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

        if (Direction == 1)
        LeftImplRight = -1;
        else
        RightImplLeft = -1;

        Direction++;

        break;
    }
}
#endif
CutPoint = (GNL_EC_CUT_POINT)BListElt (G_ListAllCutPoints, Index);
VarBddFct = GnlEcCutPointBddFct (CutPoint);

/*
if ((Direction == 2) && (Index == 100))

```

```

{
    ResetTagCountBit (BddImpl);
    IndexExists = SearchForIndexInBdd (BddImpl, 100);
    ResetTagCountBit (BddImpl);

    TagOccurBitExists = SearchForTagOccurBit (BddImpl);
    ResetTagCountBit (BddImpl);

    TotalResetTagOccurBit (BddImpl);
    ResetTagCountBit (BddImpl);
}
*/
if (bdd_compose (BddImpl, Index, VarBddFct, &BddRes))
{
#endif PRINT_COMPOSE_STAT
    fprintf (stderr, "Tried to compose Var = %s (NbNodes = %d) at phase
%d\n",
            BddVarName (GnlEcCutPointBddVar (CutPoint)),
            GnlEcCutPointNbNode (CutPoint), Phase);
#endif
#endif LEAST_BDD_HEUR
    if (Phase == 0)
    {
#endif COMPOSE_UNTIL_FIRST_FAILURE
        SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
        SameBddXor = 1;
        continue;
#else
        /* We try the second heuristic. */ */
        Phase = 2;
#endif
#endif
    }

#endif COMMON_CUTPOINTS_HEUR
    if (Phase == 1)
    {
        Phase = 2;
        break;
    }
#endif
else
{
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

    if (Direction == 1)
    LeftImplRight = -1;
    else
    RightImplLeft = -1;

    Direction++;

    break;
}
break;
#endif
#endif COMPOSE_UNTIL_FIRST_FAILURE

```

```

        SetGnlNodeValue (GnlEcCutPointNode (CutPoint), -1);
        SameBddImpl = 1;
        continue;
#else
        GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

        if (Direction == 1)
            LeftImplRight = -1;
        else
            RightImplLeft = -1;

        Direction++;

        break;
#endif
#endif
}
BddImpl = BddRes;
SameBddImpl = 0;

#ifndef PRINT_COMPOSE_STAT
    fprintf (stderr, "Composed Var = %s (NbNodes = %d) at phase %d\n",
            BddVarName (GnlEcCutPointBddVar (CutPoint)),
            GnlEcCutPointNbNode (CutPoint), Phase);
#endif

if (BddImpl == bdd_one ())
{
    GnlFreeWorkSpaceFromNodes (Ws, Node1, Node2);

    if (Direction == 1)
        LeftImplRight = 1;
    else
        RightImplLeft = 1;

    Direction++;

    break;
}
}

BListDelete (&G_ListAllCutPoints, GnlCutPointFree);
#ifndef COMMON_CUTPOINTS_HEUR
    BListQuickDelete (&G_ListOut1CutPoints);
    BListQuickDelete (&G_ListOut2CutPoints);
    BListQuickDelete (&G_ListCommonCutPoints);
#endif
}

/*
 *Invert = -1;
*/
if ((LeftImplRight > 0) && (RightImplLeft > 0))
{
    if (ReconnectNode (SecondNode, FirstNode, Invert))
        return (GNL_MEMORY_FULL);
}

```

```

        return (GNL_EQUIVALENT_PHASE0 + Phase);
    }

/*
if (LeftImplRight > 0)
{
    if (ReconnectToORNNode (RefGnl, FirstNode, SecondNode))
        return (GNL_MEMORY_FULL);

    *Invert = 0;
}

if (RightImplLeft > 0)
{
    if (ReconnectToORNNode (RefGnl, SecondNode, FirstNode))
        return (GNL_MEMORY_FULL);

    *Invert = 1;
}
*/
if (!LeftImplRight || !RightImplLeft)
    return (GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS + Phase);

return (GNL_UNRESOLVED);
}
#endif

/*
 * GnlResolveFalseNegative
 */
/* -----
GNL_STATUS GnlResolveFalseNegative (Gnl1, Gnl2, GnlMayFalse,
                                    GnlErrors, GnlBadPatterns)
{
    GNL                      Gnl1;
    GNL                      Gnl2;
    BLIST                     GnlMayFalse;
    BLIST                     GnlErrors;
    BLIST                     GnlBadPatterns;
    int                       i;
    int                       Ind;
    int                       Invert;
    GNL_NODE                  Out1;
    GNL_NODE                  Out2;
    BDD_STATUS                Status;
    int                       Phase;
    BLIST                     BadPattern;

    for (i=0; i<BListSize (GnlMayFalse); i++)
    {
        Ind = BListElt (GnlMayFalse, i);
        Out1 = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
        Out2 = (GNL_NODE)BListElt (G_OutputNodes2, Ind);

        Status = GnlResolveTwoNodes (200000, Gnl1, Out1, Out2, &Invert);
    }
}
*/

```

```

if (Status == GNL_MEMORY_FULL)
return (GNL_MEMORY_FULL);

if (Status == GNL_UNRESOLVED_AT_XOR_BUILD)
{
    fprintf (stderr, "    o [%s] Unresolved at XOR-building phase !\n",
             GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
    continue;
}

if (Status == GNL_EQUIVALENT_AT_XOR_BUILD)
{
    fprintf (stderr, "    o [%s] Resolved at XOR-building phase !\n",
             GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
    BListDelInsert (GnlMayFalse, i+1);
    i--;
    continue;
}

if (Status == GNL_ERROR_DETECTED_AT_XOR_BUILD)
{
    fprintf (stderr, "    o [%s] Error Detected at XOR-building phase
!!!!\n",
             GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));

    BListAddElt (GnlBadPatterns, -1);

    if (BListAddElt (GnlErrors, Ind))
        return (GNL_MEMORY_FULL);
    BListDelInsert (GnlMayFalse, i+1);
    i--;
    continue;
}

if (Status == GNL_UNRESOLVED_AT_BUILD_CUTPOINT)
{
    fprintf (stderr, "    o [%s] Unresolved at BuildCutPoint phase !\n",
             GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
    continue;
}

if ((Status >= GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS) &&
    (Status <= GNL_ERROR_DETECTED_PHASE2_ONLY_INPUTS))
{
    Phase = Status - GNL_ERROR_DETECTED_PHASE0_ONLY_INPUTS;
    fprintf (stderr,
            "    o [%s] Error Detected at phase %d - only input vars !!!\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)), Phase);

    if (G_BadPatternsFound == 1)
    {
        if (BListCopyNoEltCr ((BLIST)BListElt (G_ListBadPatterns,
                                                BListSize (G_ListBadPatterns) - 1),
                               &BadPattern))
            return (GNL_MEMORY_FULL);
        BListAddElt (GnlBadPatterns, (int)BadPattern);
    }
}

```

```

else
    BListAddElt (GnlBadPatterns, G_BadPatternsFound);

if (BListAddElt (GnlErrors, Ind))
    return (GNL_MEMORY_FULL);
BListDelInsert (GnlMayFalse, i+1);
i--;
continue;
}

if (Status == GNL_UNRESOLVED_AT_EXTRACTION)
{
#endif LEAST_BDD_HEUR
    fprintf (stderr, " o [%s] Unresolved at 2-d Extract phase !\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
#else
    fprintf (stderr, " o [%s] Unresolved at 0-th Extract phase !\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
#endif
    continue;
}

if (Status == GNL_UNRESOLVED)
{
#endif LEAST_BDD_HEUR
    fprintf (stderr, " o [%s] Unresolved at phase 2 !\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
#else
    fprintf (stderr, " o [%s] Unresolved at phase 0 !\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
#endif
    continue;
}

if ((Status >= GNL_EQUIVALENT_PHASE0) &&
    (Status <= GNL_EQUIVALENT_PHASE2))
{
    Phase = Status - GNL_EQUIVALENT_PHASE0;
    fprintf (stderr, " o [%s] Resolved at phase %d !\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)), Phase);
    BListDelInsert (GnlMayFalse, i+1);
    i--;
    continue;
}

if ((Status >= GNL_ERROR_DETECTED_PHASE0) &&
    (Status <= GNL_ERROR_DETECTED_PHASE2))
{
    Phase = Status - GNL_ERROR_DETECTED_PHASE0;
    fprintf (stderr, " o [%s] Error Detected at phase %d !!!\n",
            GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)), Phase);

    BListAddElt (GnlBadPatterns, -1);

    if (BListAddElt (GnlErrors, Ind))
        return (GNL_MEMORY_FULL);
}

```

gnlec.c

```
    BListDelInsert (GnlMayFalse, i+1);
    i--;
    continue;
}
}

return (GNL_OK);
}

#ifndef RANDOM_SIMULATION
/*-----*/
/* PrepareRandomInputVectors */
/*-----*/
GNL_STATUS PrepareRandomInputVectors (ListIn)
    BLIST           ListIn;
{
    int             NbIn = BListSize (ListIn);
    KEY_TYPE       *Seed;
    int             i;
    int             j;
    int             NumBadPatterns = 0;
    int             NewInd;
    GNL_CONSTRAINT NewConstraint;
    BLIST           NewInVector;
    GNL_NODE        CurrInNode;
    BLIST           TmpList;

    srand (G_SimulationNum);

    if (BListCreateWithSize (RANDOM_IN_VECTORS_NUM, &G_InVectors))
        return (GNL_MEMORY_FULL);

    if (G_ListBadPatterns && BListSize (G_ListBadPatterns))
    {
        if ((NumBadPatterns = BListSize (G_ListBadPatterns)) >
RANDOM_IN_VECTORS_NUM)
        {
            for (i=0; i<RANDOM_IN_VECTORS_NUM; i++)
            {
                NewInd = (int) (NumBadPatterns * rand ()/(RAND_MAX + 1.0));
                BListAddElt (G_InVectors, BListElt (G_ListBadPatterns, NewInd));
                BListDelInsert (G_ListBadPatterns, NewInd+1);
                NumBadPatterns--;
            }
            return (GNL_OK);
        }
    }
    else
    {
        TmpList = G_InVectors;
        G_InVectors = G_ListBadPatterns;
        G_ListBadPatterns = TmpList;
        if (NumBadPatterns == RANDOM_IN_VECTORS_NUM)
            return (GNL_OK);
    }
}

if (!(Seed = (KEY_TYPE *)calloc (NbIn, sizeof (KEY_TYPE))))
```

```

    return (GNL_MEMORY_FULL);

for (i=NumBadPatterns; i<RANDOM_IN_VECTORS_NUM; i++)
{
    if (BListCreateWithSize (NbIn, &NewInVector))
        return (GNL_MEMORY_FULL);
    BListAddElt (G_InVectors, (int)NewInVector);
}

for (i=0; i<NbIn; i++)
    Seed[i] = rand ();

for (i=0; i<NbIn; i++)
{
    CurrInNode = (GNL_NODE)GnlVarDads ((GNL_VAR)BListElt (ListIn, i));
    if (!CurrInNode)
        continue;
    srand (Seed[i]);
    for (j=NumBadPatterns; j<RANDOM_IN_VECTORS_NUM; j++)
    {
        NewConstraint = (rand ()/(RAND_MAX+1.0) < 0.5 ?
                         GnlNodeNegConstraint (CurrInNode) :
                         GnlNodePosConstraint (CurrInNode));
        BListAddElt ((BLIST)BListElt (G_InVectors, j), (int)NewConstraint);
    }
}

free (Seed);

return (GNL_OK);
}

/*-----
/* PrintVectors
/*-----
void PrintVectors (void)
{
    int          i;
    int          j;
    BLIST        VectorI;
    GNL_CONSTRAINT ConstraintI;

    for (i=0; i<RANDOM_IN_VECTORS_NUM; i++)
    {
        VectorI = (BLIST)BListElt (G_InVectors, i);
        for (j=0; j<BListSize (VectorI); j++)
        {
            ConstraintI = (GNL_CONSTRAINT)BListElt (VectorI, j);
            printf ("%c", (ConstraintI ?
                           VALUE (GnlConstraintValue (ConstraintI)) : VALUE (-1)));
        }
        printf ("\n");
    }
}

/*-----
/* RandomSimulation
*/

```

gnlec.c

```
-----*/
GNL_STATUS RandomSimulation (Gn11, Gn12, OutList)
    GNL          Gn11;
    GNL          Gn12;
    BLIST        OutList; /* List of output indexes           */
{                                         */
    int          i;
    int          j;
    int          Ind;

    for (i=0; i<BListSize (G_InVectors); i++)
    {
        ResetInputs (Gn11);
        SetInputs ((BLIST)BListElt (G_InVectors, i));

        MakeNewTag (Gn11, Gn12);

        for (j=0; j<BListSize (OutList); j++)
        {
            Ind = BListElt (OutList, j);
            if (SimulateGnlValue (Gn11, (GNL_NODE)BListElt (G_OutputNodes1, Ind),
i))
                return (GNL_MEMORY_FULL);
            if (SimulateGnlValue (Gn12, (GNL_NODE)BListElt (G_OutputNodes2, Ind),
i))
                return (GNL_MEMORY_FULL);
        }
    }

    return (GNL_OK);
}

-----*/
/* Hash
*/
/* This hash function gets as parameter the results of simulation
/* by RANDOM_IN_VECTORS_NUM random input vectors in some node and
/* returns unsigned integer value which is the perfect hashing.
*/
KEY_TYPE Hash (Signature)
    BLIST      Signature;
{
    int          i;
    KEY_TYPE Result = 0;

    for (i=0; i<BListSize (Signature); i++)
        if (BListElt (Signature, i))
            Result += pow (2, i);

    return (Result);
}

-----*/
/* KeyCmp
*/
int KeyCmp (Ptr1, Ptr2)
    BLIST      *Ptr1;
```

```

    BLIST      *Ptr2;
{
    int        i;
    KEY_TYPE  Key1;
    KEY_TYPE  Key2;

    for (i=0; i<BListSize (*Ptr1); i++)
    {
        Key1 = (KEY_TYPE)BListElt (*Ptr1, i);
        Key2 = (KEY_TYPE)BListElt (*Ptr2, i);

        if (Key1 < Key2)
            return (-1);

        if (Key1 > Key2)
            return (1);
    }

    return (0);
}

/*-----*/
/* SortNodes */
/*-----*/
GNL_STATUS SortNodes (Gnl, Node)
    GNL        Gnl;
    GNL_NODE  Node;
{
    int        i;
    int        Status;
    BLIST      *KeyPtr;
    KEY_TYPE  KeyI;
    BLIST      *NodesListPtr;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return (GNL_OK);
    SetGnlNodeTag (Node, GnlTag (Gnl));

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        if (SortNodes (Gnl, (GNL_NODE)BListElt (GnlNodeSons (Node), i)))
            return (GNL_MEMORY_FULL);

    if (!(KeyPtr = (BLIST *)calloc (1, sizeof (BLIST))))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (BListSize (GnlNodeSignatures (Node)), KeyPtr))
        return (GNL_MEMORY_FULL);
    for (i=0; i<BListSize (GnlNodeSignatures (Node)); i++)
    {
        KeyI = Hash ((BLIST)BListElt (GnlNodeSignatures (Node), i));
        BListAddElt (*KeyPtr, (int)KeyI);
    }

    if (NodesListPtr = (BLIST *)SkipListFindData (G_PossiblyEquiv, KeyPtr))

```

```

{
    if (BListAddElt (*NodesListPtr, (int)Node))
        return (GNL_MEMORY_FULL);
    SetGnlNodeEquivList (Node, *NodesListPtr);
    BListQuickDelete (KeyPtr);
    free (KeyPtr);
    return (GNL_OK);
}

if (!(NodesListPtr = (BLIST *)calloc (1, sizeof (BLIST))))
    return (GNL_MEMORY_FULL);

if (BListCreate (NodesListPtr))
    return (GNL_MEMORY_FULL);
BListAddElt (*NodesListPtr, (int)Node);
SetGnlNodeEquivList (Node, *NodesListPtr);

if (Status = SkipListInsert (G_PossiblyEquiv, KeyPtr, NodesListPtr))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* MarkFanOut
/*-----*/
void MarkFanOut (Node)
    GNL_NODE      Node;
{
    int          i;

    if (GnlNodeEquivTag (Node) == G_SimulationNum)
        return;

    SetGnlNodeEquivTag (Node, G_SimulationNum);

    if (GnlNodeDads (Node))
        for (i=0; i<BListSize (GnlNodeDads (Node)); i++)
            MarkFanOut ((GNL_NODE)BListElt (GnlNodeDads (Node), i));
}

/*-----
/* MatchInternalWithBdds
/*-----*/
GNL_STATUS MatchInternalWithBdds (RefGnl, ListNodes,
                                    GnlMayFalse, GnlErrors, GnlBadPatterns)
    GNL          RefGnl;
    BLIST        ListNodes;
    BLIST        GnlMayFalse;
    BLIST        GnlErrors;
    BLIST        GnlBadPatterns;
{
    int          i;
    int          j;
    int          k;
    int          Ind;
    GNL_NODE    OutNode1;
}

```

gnlec.c

```
GNL_NODE      OutNode2;
GNL_NODE      Node1;
GNL_NODE      Node2;
int          Status;
int          Invert = 0;
BLIST        BadPattern;

for (i=0; i<BListSize (ListNodes); i++)
{
    for (j=i+1; j<BListSize (ListNodes); j++)
    {
        Node1 = (GNL_NODE)BListElt (ListNodes, i);
        Node2 = (GNL_NODE)BListElt (ListNodes, j);

        if ((GnlNodeEquivTag (Node1) < G_SimulationNum - 1) &&
            (GnlNodeEquivTag (Node2) < G_SimulationNum - 1))
            continue;

        G_NbEquiv++;

        Status = GnlResolveTwoNodes (100000, RefGnl, Node1, Node2, &Invert);

        if ((Status >= GNL_EQUIVALENT) &&
            (Status <= GNL_EQUIVALENT_PHASE2))
        {
            fprintf (stderr, "+");

            if (Invert)
                BListElt (ListNodes, i) = (int)Node2;

            G_EquivFound++;

            MarkFanOut ((GNL_NODE)BListElt (ListNodes, i));

            BListDelInsert (ListNodes, j+1);
            j--;
        }

        if ((Status >= GNL_ERROR_DETECTED) &&
            (Status <= GNL_ERROR_DETECTED_AT_XOR_BUILD))
        {
            G_DiffFound++;
            /*
            if (Invert == 0)
            {
                G_ImplFound++;
                MarkFanOut (Node1);
                fprintf (stderr, ">");
            }

            if (Invert == 1)
            {
                G_ImplFound++;
                MarkFanOut (Node2);
                fprintf (stderr, "<");
            }
        }
    }
}
```

```

if (Invert == -1)
    fprintf (stderr, "-");
*/
fprintf (stderr, "-");
}

if (Status <= GNL_UNRESOLVED_AT_EXTRACTION)
{
/*
if (Invert == 0)
{
    G_ImplFound++;
    MarkFanOut (Node1);
    fprintf (stderr, "l");
}

if (Invert == 1)
{
    G_ImplFound++;
    MarkFanOut (Node2);
    fprintf (stderr, "r");
}

if (Invert == -1)
    fprintf (stderr, "?");
*/
fprintf (stderr, "?");
}
else
{
    for (k=0; k<BListSize (GnlMayFalse); k++)
    {
        Ind = BListElt (GnlMayFalse, k);
        OutNode1 = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
        OutNode2 = (GNL_NODE)BListElt (G_OutputNodes2, Ind);

        if ((Status >= GNL_EQUIVALENT) &&
            (Status <= GNL_EQUIVALENT_PHASE2))
        {
            if (OutNode1 == OutNode2)
                BListDelInsert (GnlMayFalse, k+1);
            continue;
        }

        if (((Node1 == OutNode1) && (Node2 == OutNode2)) ||
            ((Node1 == OutNode2) && (Node2 == OutNode1)))
        {
            if (G_BadPatternsFound == 1)
            {
                if (BListCopyNoEltCr ((BLIST)BListElt (G_ListBadPatterns,
                                                BListSize (G_ListBadPatterns) -
                                                1),
                                         &BadPattern))
                    return (GNL_MEMORY_FULL);
                BListAddElt (GnlBadPatterns, (int)BadPattern);
            }
        }
    }
}

```

```

        }
    else
        BListAddElt (GnlBadPatterns, G_BadPatternsFound);

    if (BListAddElt (GnlErrors, Ind))
        return (GNL_MEMORY_FULL);

    BListDelInsert (GnlMayFalse, k+1);
    continue;
}
}

if (!BListSize (GnlMayFalse))
{
    i = BListSize (ListNodes);
    break;
}

SetGnlNodeEquivTag (Node1, G_SimulationNum - 2);
SetGnlNodeEquivTag (Node2, G_SimulationNum - 2);
}
}

return (GNL_OK);
}

/*-----*/
/* InsertIntoReadyList */
/*-----*/
GNL_STATUS InsertIntoReadyList (Gnl, ReadyList, Node)
    GNL        Gnl;
    BLIST     *ReadyList;
    GNL_NODE Node;
{
    int        i;
    BLIST     NodesEquivList;
    GNL_NODE FirstNode;
    GNL_NODE Node1;

    if (GnlNodeTag (Node) == GnlTag (Gnl))
        return (GNL_OK);
    SetGnlNodeTag (Node, GnlTag (Gnl));

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        if (InsertIntoReadyList (Gnl, ReadyList, (GNL_NODE)BListElt (GnlNodeSons
(Node), i)))
            return (GNL_MEMORY_FULL);

    NodesEquivList = GnlNodeEquivList (Node);
    FirstNode = (GNL_NODE)BListElt (NodesEquivList, 0);

    if (GnlNodeTagBdd (FirstNode) == GnlTagBdd (Gnl))

```

```

    return (GNL_OK);
SetGnlNodeTagBdd (FirstNode, GnlTagBdd (Gnl));

#ifndef PRINT_SIMULATION_BUCKETS
if (BListSize (NodesEquivList) > 1)
{
    fprintf (stderr, "List of %d nodes:\n", BListSize (NodesEquivList));
    for (i=0; i<BListSize (NodesEquivList); i++)
    {
        NodeI = (GNL_NODE)BListElt (NodesEquivList, i);
        fprintf (stderr, " %s", (GnlNodeVar (NodeI)) ?
                    GnlVarName (GnlNodeVar (NodeI)) : "No var"));
    }
    fprintf (stderr, "\n");
}
#endif

if (BListSize (NodesEquivList) < 2)
    return (GNL_OK);

if (BListSize (NodesEquivList) > MAX_CHECKED_LIST_SIZE)
{
    for (i=0; i<BListSize (NodesEquivList); i++)
        SetGnlNodeEquivTag ((GNL_NODE)BListElt (NodesEquivList, i),
G_SimulationNum);
    return (GNL_OK);
}

return (BListAddElt (ReadyList[GnlNodeHeight (FirstNode)-1],
(int)NodesEquivList));
}

/*-----
/* EquivCheckGnlWithRandomSimulation
/*-----*/
GNL_STATUS EquivCheckGnlWithRandomSimulation (Gnl1, Gnl2, GnlMayFalse,
                                              GnlErrors, GnlBadPatterns)
{
    GNL      Gnl1;
    GNL      Gnl2;
    BLIST     GnlMayFalse;
    BLIST     GnlErrors;
    BLIST     GnlBadPatterns;
{
    int      i;
    int      j;
    int      NbUnchange = 1;
    int      Ind;
    GNL_NODE Out1;
    GNL_NODE Out2;
    KEY_TYPE *CurrKey;
    BLIST   *CurrNodesListPtr;
    BLIST   CurrSignature1;
    BLIST   CurrSignature2;
    BLIST   NewBadPattern;
    SKIPLIST PrevPossiblyEquiv;
    int      MaxHeight;
    GNL_NODE OutNodeI;
}

```

```

BLIST      CurrList;
BLIST      *ReadyList;

do
{
    G_SimulationNum++;
    G_NbEquiv = G_DiffFound = G_EquivFound = 0;

    if (PrepareRandomInputVectors (GnlRelevantInputs (Gnl1)))
        return (GNL_MEMORY_FULL);

    if (RandomSimulation (Gnl1, Gnl2, GnlMayFalse))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlMayFalse); i++)
    {
        Ind = BListElt (GnlMayFalse, i);
        Out1 = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
        Out2 = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
        CurrSignature1 = (BLIST)BListElt (GnlNodeSignatures (Out1),
G_SimulationNum-1);
        CurrSignature2 = (BLIST)BListElt (GnlNodeSignatures (Out2),
G_SimulationNum-1);
        if (Hash (CurrSignature1) != Hash (CurrSignature2))
        {
            fprintf (stderr, "    o [%s] Error Detected - different signatures
!!!!\n",
                    GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind)));
            if (BListAddElt (GnlErrors, Ind))
                return (GNL_MEMORY_FULL);
            for (j=0; j<BListSize (CurrSignature1); j++)
                if (BListElt (CurrSignature1, j) != BListElt (CurrSignature2, j))
                    break;
            if (BListCopyNoEltCr ((BLIST)BListElt (G_InVectors, j),
&NewBadPattern))
                return (GNL_MEMORY_FULL);
            BListAddElt (GnlBadPatterns, NewBadPattern);
            BListDelInsert (GnlMayFalse, i+1);
            i--;
        }
    }

    BListDelete (&G_InVectors, BListQuickDelete);

    if (!BListSize (GnlMayFalse))
        return (GNL_OK);

    PrevPossiblyEquiv = G_PossiblyEquiv;

    if (SkipListCreate (&G_PossiblyEquiv, KeyCmp))
        return (GNL_MEMORY_FULL);

    MakeNewTag (Gnl1, Gnl2);

    for (i=0; i<BListSize (GnlMayFalse); i++)
    {
        Ind = BListElt (GnlMayFalse, i);

```

```

if (SortNodes (Gn11, (GNL_NODE)BListElt (G_OutputNodes1, Ind)))
    return (GNL_MEMORY_FULL);
if (SortNodes (Gn11, (GNL_NODE)BListElt (G_OutputNodes2, Ind)))
    return (GNL_MEMORY_FULL);
}

if ((G_SimulationNum > 1) &&
    (SkipListSize (G_PossiblyEquiv) == SkipListSize (PrevPossiblyEquiv)))
    NbUnchange++;
else
    NbUnchange = 1;

MakeNewTag (Gn11, Gn12);

MaxHeight = 0;

for (i=0; i<BListSize (GnlMayFalse); i++)
{
    Ind = BListElt (GnlMayFalse, i);

    OutNodeI = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
    ComputeNodesHeight (Gn11, OutNodeI);
    if (MaxHeight < GnlNodeHeight (OutNodeI))
        MaxHeight = GnlNodeHeight (OutNodeI);

    OutNodeI = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
    ComputeNodesHeight (Gn12, OutNodeI);
    if (MaxHeight < GnlNodeHeight (OutNodeI))
        MaxHeight = GnlNodeHeight (OutNodeI);
}

if (CreateAllLayers (MaxHeight, &ReadyList))
    return (GNL_MEMORY_FULL);

MakeNewTag (Gn11, Gn12);
MakeNewTagBdd (Gn11, Gn12);

for (i=0; i<BListSize (GnlMayFalse); i++)
{
    Ind = BListElt (GnlMayFalse, i);
    OutNodeI = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
    if (InsertIntoReadyList (Gn11, ReadyList, OutNodeI))
        return (GNL_MEMORY_FULL);
    OutNodeI = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
    if (InsertIntoReadyList (Gn12, ReadyList, OutNodeI))
        return (GNL_MEMORY_FULL);
}

fprintf (stderr, "\n");

for (i=0; i<MaxHeight; i++)
{
    for (j=0; j<BListSize (ReadyList[i]); j++)
    {
        CurrList = (BLIST)BListElt (ReadyList[i], j);
        if (MatchInternalWithBdds (Gn11, CurrList,
                                   GnlMayFalse, GnlErrors, GnlBadPatterns))

```

gnlec.c

```
        return (GNL_MEMORY_FULL);

    if (!BListSize (GnlMayFalse))
    {
        i = MaxHeight;
        break;
    }
}

fprintf (stderr, "\n");

for (i=0; i<MaxHeight; i++)
    BListQuickDelete (&(ReadyList[i]));

free (ReadyList);

fprintf (stderr,
        "\n      Simulation %d: size - %d, checked - %d, equiv. - %d,
diff. - %d\n",
        G_SimulationNum, SkipListSize (G_PossiblyEquiv),
        G_NbEquiv, G_EquivFound, G_DiffFound);

SkipListDelete (&PrevPossiblyEquiv, BListQuickDelete, BListQuickDelete);
}
while (BListSize (GnlMayFalse) &&
       (G_EquivFound ||
        (G_ListBadPatterns && BListSize (G_ListBadPatterns)) ||
        (NbUnchange < 10)));

SkipListDelete (&G_PossiblyEquiv, BListQuickDelete, BListQuickDelete);

return (GNL_OK);
}
#endif

/*-----*/
/* PrintStatistics */
/*-----*/
void PrintStatistics (Num_Equiv, Num_Err, Num_Unproved)
{
    int          Num_Equiv;
    int          Num_Err;
    int          Num_Unproved;
{
    fprintf (stderr, " Nb Functions proved equivalent = %d\n", Num_Equiv);
    fprintf (stderr, " Nb Functions proved non-equivalent = %d\n", Num_Err);
    fprintf (stderr, " Nb Functions unproved = %d\n", Num_Unproved);
    fprintf (stderr, " MaxNb Nodes = %d\n", G_GnlMaxBddNode);
    fprintf (stderr, " Nb CutVar = %d\n", G_NbCutVar);
    fprintf (stderr, " Nb Bridge Nodes = %d\n", G_BridgeNodes);
    fprintf (stderr, " Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);
    fprintf (stderr, "\n");
}

/*-----*/
/* PrintBadPattern */
/*-----*/
```

gnlec.c

```
void PrintBadPattern (Pattern)
    int          Pattern;
{
    int          i;
    BLIST        BadPattern;
    GNL_CONSTRAINT ConstrainI;
    GNL_NODE     NodeI;
    int          Value;

    fprintf (stderr, "[");
    if (Pattern == -1)
        fprintf (stderr, " Always different!");
    else
    {
        BadPattern = (BLIST) Pattern;
        ConstrainI = (GNL_CONSTRAINT) BListElt (BadPattern, 0);
        NodeI = GnlConstraintNode (ConstrainI);
        Value = GnlConstraintValue (ConstrainI);
        fprintf (stderr, " %s = %c \n",
                 GnlVarName (GnlNodeVar (NodeI)), VALUE (Value));
        for (i=1; i<BListSize (BadPattern)-1; i++)
        {
            ConstrainI = (GNL_CONSTRAINT) BListElt (BadPattern, i);
            NodeI = GnlConstraintNode (ConstrainI);
            Value = GnlConstraintValue (ConstrainI);
            fprintf (stderr,
                     " %s = %c \n",
                     GnlVarName (GnlNodeVar (NodeI)), VALUE (Value));
        }
        ConstrainI = (GNL_CONSTRAINT) BListElt (BadPattern, i);
        NodeI = GnlConstraintNode (ConstrainI);
        Value = GnlConstraintValue (ConstrainI);
        fprintf (stderr,
                 " %s = %c ]",
                 GnlVarName (GnlNodeVar (NodeI)), VALUE (Value));
    }
}

/*-----
/* PrintResults
/*-----*/
void PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns)
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST        OutMayFalse;
    BLIST        OutErrors;
    BLIST        ListBadPatterns;
{
    int          i;
    int          Ind;
    int          NbMayFalse = BListSize (OutMayFalse);
    int          NbErrors = BListSize (OutErrors);
    GNL_NODE    OutNode1;
    GNL_NODE    OutNode2;
    char         *Namel;
```

```

char          *Name2;

fprintf (stderr, "\n");
fprintf (stderr, " Output(s) proved different: %d\n", NbErrors);
fprintf (stderr, " -----\n");
for (i=0; i<NbErrors; i++)
{
    Ind = BListElt (OutErrors, i);
    Name1 = GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind));
    Name2 = GnlVarName ((GNL_VAR)BListElt (G_Outputs2, Ind));

    fprintf (stderr, "\n <%s> != <%s>", Name1, Name2);

    if (ListBadPatterns && BListElt (ListBadPatterns, i))
    {
        fprintf (stderr, "\n           Counter-Example Vector = ");
        PrintBadPattern (BListElt (ListBadPatterns, i));

        if (BListElt (ListBadPatterns, i) != -1)
        {
            OutNode1 = (GNL_NODE)BListElt (G_OutputNodes1, Ind);
            OutNode2 = (GNL_NODE)BListElt (G_OutputNodes2, Ind);
            MakeNewTag (Gnl1, Gnl2);
            ResetInputs (Gnl1);
            SetInputs ((BLIST)BListElt (ListBadPatterns, i));
            SimulateGnlValue (Gnl1, OutNode1, -1);
            fprintf (stderr, "\n           Function: %s = %c", Name1,
                     VALUE (GnlNodeValue (OutNode1)));

            SimulateGnlValue (Gnl2, OutNode2, -1);
            fprintf (stderr, "\n           Function: %s = %c", Name2,
                     VALUE (GnlNodeValue (OutNode2)));
        }
    }
    fprintf (stderr, "\n");
}

fprintf (stderr, "\n");
fprintf (stderr, " Output(s) not proved: %d\n", NbMayFalse);
fprintf (stderr, " -----\n");
for (i=0; i<NbMayFalse; i++)
{
    Ind = BListElt (OutMayFalse, i);
    Name1 = GnlVarName ((GNL_VAR)BListElt (G_Outputs1, Ind));
    Name2 = GnlVarName ((GNL_VAR)BListElt (G_Outputs2, Ind));

    fprintf (stderr, "\n <%s> != <%s>", Name1, Name2);

    fprintf (stderr, "\n");
}

BListDelete (&G_ListBadPatterns, BListQuickDelete);

FreeHookFields (Gnl1, Gnl2);

BListQuickDelete (&G_Outputs1);
BListQuickDelete (&G_Outputs2);

```

```

BListQuickDelete (&G_OutputNodes1);
BListQuickDelete (&G_OutputNodes2);

BListQuickDelete (&GnlRelevantInputs (Gnl1));
BListQuickDelete (&GnlRelevantInputs (Gnl2));
BListQuickDelete (&GnlVirtualOutputs (Gnl1));
BListQuickDelete (&GnlVirtualOutputs (Gnl2));
free (GnlHook (Gnl1));
free (GnlHook (Gnl2));
}

/*-----
/* ResetNodesFields
/*-----*/
void ResetNodesFields (Node)
    GNL_NODE      Node;
{
    int          i;

    if (GnlNodeTag (Node) == G_Tag)
        return;
    SetGnlNodeTag (Node, G_Tag);

    SetGnlNodeValue (Node, 0);
    SetGnlNodeTagBdd (Node, 0);
    SetGnlNodeEquivTag (Node, 0);

    if ((GnlNodeOp (Node) == GNL_CONSTANTE) ||
        (GnlNodeOp (Node) == GNL_VARIABLE))
        return;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        ResetNodesFields ((GNL_NODE)BListEl (GnlNodeSons (Node), i));
}

/*-----
/* GnlEquivCheckGnl
/*-----*/
GNL_STATUS GnlEquivCheckGnl (Gnl1, Gnl2, ListAssocIn, ListAssocOut,
                             MaxNbNodes, NbMayErrors, GnlMayErrors,
                             NbErrors, GnlErrors, GnlBadPatterns)
{
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST        ListAssocIn;
    BLIST        ListAssocOut;
    int          MaxNbNodes;
    int          *NbMayErrors;
    BLIST        *GnlMayErrors;
    int          *NbErrors;
    BLIST        *GnlErrors;
    BLIST        *GnlBadPatterns;
{
    int          i;
    int          NbIn;
    int          NbOut;
    BLIST        NewList;
    int          NbMayFalse;
}

```

gnlec.c

```
int      NbEquiv;
BLIST   OutMayFalse = NULL;
int      NbCurrentError;
BLIST   OutErrors = NULL;
BLIST   ListBadPatterns = NULL; /* BLIST of BLISTS of GNL_NODES      */
void     *Ptr;

fprintf (stderr, "\n");
fprintf (stderr, "    o Verification started ... \n");

if (BListSize (ListAssocIn) % 2)
{
    fprintf (stderr, "CMPGNL-ERROR: netlists have different number of
inputs\n");
    return (GNL_MEMORY_FULL);
}

if (BListSize (ListAssocOut) % 2)
{
    fprintf (stderr, "CMPGNL-ERROR: netlists have different number of
outputs\n");
    return (GNL_MEMORY_FULL);
}

NbIn = BListSize (ListAssocIn) / 2;
NbOut = BListSize (ListAssocOut) / 2;

if ((Ptr = calloc (1, sizeof (ADD_GNL_REC))) == NULL)
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlHook (Gnl1, Ptr);

if ((Ptr = calloc (1, sizeof (ADD_GNL_REC))) == NULL)
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlHook (Gnl2, Ptr);

if (BListCreateWithSize (NbIn, &NewList))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlRelevantInputs (Gnl1, NewList);

if (BListCreateWithSize (NbIn, &NewList))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlRelevantInputs (Gnl2, NewList);

if (BListCreateWithSize (NbOut, &NewList))
```

```

{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlVirtualOutputs (Gnl1, NewList);

if (BListCreateWithSize (NbOut, &NewList))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
SetGnlVirtualOutputs (Gnl2, NewList);

if (BListCreateWithSize (NbOut, &G_OutputNodes1))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

if (BListCreateWithSize (NbOut, &G_OutputNodes2))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (ListAssocOut); i++)
    SetGnlVarDir ((GNL_VAR)BListElt (ListAssocOut, i), GNL_VAR_OUTPUT);

for (i=0; i<BListSize (ListAssocIn); i++)
    SetGnlVarDir ((GNL_VAR)BListElt (ListAssocIn, i), GNL_VAR_INPUT);

for (i=0; i<NbIn; i++)
{
    BListAddElt (GnlRelevantInputs (Gnl1), BListElt (ListAssocIn, 2*i));
    BListAddElt (GnlRelevantInputs (Gnl2), BListElt (ListAssocIn, 2*i + 1));
}

for (i=0; i<NbOut; i++)
{
    BListAddElt (GnlVirtualOutputs (Gnl1), BListElt (ListAssocOut, 2*i));
    BListAddElt (GnlVirtualOutputs (Gnl2), BListElt (ListAssocOut, 2*i + 1));
}

if (BListCopyNoEltCr (GnlVirtualOutputs (Gnl1), &G_Outputs1))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

if (BListCopyNoEltCr (GnlVirtualOutputs (Gnl2), &G_Outputs2))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

```

```

/*-----*/
for (i=0; i<BListSize (GnlInputs (Gnl1)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlInputs (Gnl1), i), NULL);

for (i=0; i<BListSize (GnlInputs (Gnl2)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlInputs (Gnl2), i), NULL);

for (i=0; i<BListSize (GnlLocals (Gnl1)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlLocals (Gnl1), i), NULL);

for (i=0; i<BListSize (GnlLocals (Gnl2)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlLocals (Gnl2), i), NULL);

for (i=0; i<BListSize (GnlOutputs (Gnl1)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlOutputs (Gnl1), i), NULL);

for (i=0; i<BListSize (GnlOutputs (Gnl2)); i++)
    SetGnlVarDads ((GNL_VAR)BListElt (GnlOutputs (Gnl2), i), NULL);
/*-----*/

/* We remove all local variables' nodes, because */  

/* they can affect drastically equivalence checking performances. */
fprintf (stderr, "\n");
fprintf (stderr, " o Preprocessing ... \n");

if (RemoveLocals (Gnl1, G_OutputNodes1))
    return (GNL_MEMORY_FULL);
if (RemoveLocals (Gnl2, G_OutputNodes2))
    return (GNL_MEMORY_FULL);

#ifndef SPLITTING_NODES
    for (i=0; i<BListSize (G_OutputNodes1); i++)
    {
        ComputeMaxInput ((GNL_NODE)BListElt (G_OutputNodes1, i));
        ComputeMaxInput ((GNL_NODE)BListElt (G_OutputNodes2, i));
    }

    G_Tag++;
    if (SplitNodesInGnl (Gnl1, G_OutputNodes1))
        return (GNL_MEMORY_FULL);
    if (SplitNodesInGnl (Gnl2, G_OutputNodes2))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (G_OutputNodes1); i++)
    {
        ResetHookFields ((GNL_NODE)BListElt (G_OutputNodes1, i));
        ResetHookFields ((GNL_NODE)BListElt (G_OutputNodes2, i));
    }
#endif

/* First of all we create the Hook fields of all Vars and Nodes */
if (CreateNodesHookFields (Gnl1, Gnl2))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

```

```

if (CreateVarsHookFields (Gnl1))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}
if (CreateVarsHookFields (Gnl2))
{
    fprintf (stderr, "CMPGNL-ERROR: cannot allocate enough memory\n");
    return (GNL_MEMORY_FULL);
}

/* Computing the Dads of each Node in the Netlists */ 
if (ComputeDadsInGnl (Gnl1, Gnl2))
    return (GNL_MEMORY_FULL);

BListQuickDelete (&GnlFunctions (Gnl1));
BListQuickDelete (&GnlFunctions (Gnl2));

/* We bind the primary inputs of the two netlists */ 
if (BindPrimaryInputs (Gnl1, Gnl2))
    return (GNL_MEMORY_FULL);

#ifndef STRUCTURAL_CHECK
/* First we try to merge structurally the maximum of local vars and */
/* nodes in the two netlists. */ 
fprintf (stderr, "\n");
fprintf (stderr, "    o Structural checking ... \n");

if (EquivCheckGnlWithStructural (Gnl1, Gnl2, &OutMayFalse))
    return (GNL_MEMORY_FULL);

/* Actually all the outputs are structurally equivalent so the job is */
/* already finished. */ 
if (!(NbMayFalse = BListSize (OutMayFalse)))
{
    *NbErrors = *NbMayErrors = 0;
    *GnlErrors = *GnlMayErrors = *GnlBadPatterns = NULL;
    fprintf (stderr, "\n");
    fprintf (stderr, "        Nb Functions proved equivalent = %d\n", NbOut);
    fprintf (stderr, "        Nb Functions proved non-equivalent = %d\n", 0);
    fprintf (stderr, "        Nb Functions unproved = %d\n", 0);
    fprintf (stderr, "        Nb Bridge Nodes = %d\n", G_BridgeNodes);
    PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);
    return (GNL_OK);
}

G_Tag++;

for (i=0; i<BListSize (G_OutputNodes1); i++)
    ResetNodesFields ((GNL_NODE)BListElt (G_OutputNodes1, i));

for (i=0; i<BListSize (G_OutputNodes2); i++)
    ResetNodesFields ((GNL_NODE)BListElt (G_OutputNodes2, i));

NbEquiv = NbOut - NbMayFalse;
fprintf (stderr, "\n");
fprintf (stderr, "        Nb Functions proved equivalent = %d\n", NbEquiv);

```

```

fprintf (stderr, "      Nb Functions unproved = %d\n", NbMayFalse);
fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
#else
    if (BListCreateWithSize (BListSize (GnlVirtualOutputs (Gnl1)),
&OutMayFalse))
        return (GNL_MEMORY_FULL);
    for (i=0; i<BListSize (GnlVirtualOutputs (Gnl1)); i++)
        BListAddElt (OutMayFalse, i);
#endif

/* Now we try to prove the equivalence of the rest of the Outputs by */
/* building their respective BDDs.                                         */

/* Bottom-up construction of local bdds.                                */
fprintf (stderr, "\n");
fprintf (stderr, "      o Local-BDD checking ... \n");

G_GnlMaxBddNode = MaxNbNodes;

if (GnlBuildBddFct (Gnl1, Gnl2, OutMayFalse))
    return (GNL_MEMORY_FULL);

GnlExtractPendingFunctions (Gnl1, Gnl2, OutMayFalse);

/* Actually all the outputs are functionally equivalent so the job is*/
/* already finished.                                                 */
if (BListSize (OutMayFalse) == 0)
{
    *NbErrors = *NbMayErrors = 0;
    *GnlErrors = *GnlMayErrors = *GnlBadPatterns = NULL;
    fprintf (stderr, "\n");
    fprintf (stderr, "      Nb Functions proved equivalent = %d\n", NbOut);
    fprintf (stderr, "      Nb Functions proved non-equivalent = %d\n", 0);
    fprintf (stderr, "      Nb Functions unproved = %d\n", 0);
    fprintf (stderr, "      MaxNb Nodes = %d\n", G_GnlMaxBddNode);
    fprintf (stderr, "      Nb CutVar = %d\n", G_NbCutVar);
    fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
    fprintf (stderr, "      Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);
    PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);
    return (GNL_OK);
}

NbEquiv = NbOut - BListSize (OutMayFalse);
fprintf (stderr, "\n");
fprintf (stderr, "      Nb Functions proved equivalent = %d\n", NbEquiv);
fprintf (stderr, "      Nb Functions proved non-equivalent = %d\n", 0);
fprintf (stderr, "      Nb Functions unproved = %d\n", BListSize
(OutMayFalse));
fprintf (stderr, "      MaxNb Nodes = %d\n", G_GnlMaxBddNode);
fprintf (stderr, "      Nb CutVar = %d\n", G_NbCutVar);
fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
fprintf (stderr, "      Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);

if (BListCreateWithSize (BListSize (OutMayFalse), &OutErrors))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (BListSize (OutMayFalse), &ListBadPatterns))
    return (GNL_MEMORY_FULL);

```

```

/* Verify the pending outputs by composing the root bdd with          */
/* break bdd variables.                                              */
#endif FND
    fprintf (stderr, "\n");
    fprintf (stderr, "    o False Negative Detection ... \n");
    if (GnlResolveFalseNegative (Gnl1, Gnl2, OutMayFalse, OutErrors,
ListBadPatterns))
        return (GNL_MEMORY_FULL);

NbMayFalse = BListSize (OutMayFalse);
NbEquiv = NbOut - NbMayFalse - BListSize (OutErrors);

if (NbMayFalse == 0)
{
    *NbMayErrors = 0;
    *NbErrors = BListSize (OutErrors);
    *GnlMayErrors = NULL;
    *GnlErrors = OutErrors;
    *GnlBadPatterns = ListBadPatterns;
    fprintf (stderr, "\n");
    fprintf (stderr, "    Nb Functions proved equivalent = %d\n",
NbEquiv);
    fprintf (stderr, "    Nb Functions proved non-equivalent = %d\n",
*NbErrors);
    fprintf (stderr, "    Nb Functions unproved = %d\n", 0);
    fprintf (stderr, "    MaxNb Nodes = %d\n", G_GnlMaxBddNode);
    fprintf (stderr, "    Nb CutVar = %d\n", G_NbCutVar);
    fprintf (stderr, "    Nb Bridge Nodes = %d\n", G_BridgeNodes);
    fprintf (stderr, "    Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);
    PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);
    return (GNL_OK);
}

fprintf (stderr, "\n");
fprintf (stderr, "    Nb Functions proved equivalent = %d\n", NbEquiv);
fprintf (stderr, "    Nb Functions proved non-equivalent = %d\n",
BListSize (OutErrors));
fprintf (stderr, "    Nb Functions unproved = %d\n", BListSize
(OutMayFalse));
fprintf (stderr, "    MaxNb Nodes = %d\n", G_GnlMaxBddNode);
fprintf (stderr, "    Nb CutVar = %d\n", G_NbCutVar);
fprintf (stderr, "    Nb Bridge Nodes = %d\n", G_BridgeNodes);
fprintf (stderr, "    Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);
#endif

#ifndef RANDOM_SIMULATION
/* If the user did not ask to ignore random simulation we call it      */
if (!GnlEnvIgnoreRandomSim ())
{
    /* Verify the pending outputs by random simulation.                      */
    fprintf (stderr, "\n");
    fprintf (stderr, "    o Random Simulation ... \n");
    if (EquivCheckGnlWithRandomSimulation (Gnl1, Gnl2, OutMayFalse,
                                                OutErrors, ListBadPatterns))
    {
        fprintf (stderr, "    Failed - cannot allocate enough memory\n");
}

```

```

    *GnlErrors = OutErrors;
    *GnlMayErrors = OutMayFalse;
    *GnlBadPatterns = ListBadPatterns;

    PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);

    return (GNL_OK);
}

NbMayFalse = BListSize (OutMayFalse);
NbEquiv = NbOut - NbMayFalse - BListSize (OutErrors);

/* Actually all the outputs are structurally equivalent so the job is*/
/* already finished. */
if (NbMayFalse == 0)
{
    *NbMayErrors = 0;
    *NbErrors = BListSize (OutErrors);
    *GnlMayErrors = NULL;
    *GnlErrors = OutErrors;
    *GnlBadPatterns = ListBadPatterns;
    fprintf (stderr, "\n");
    fprintf (stderr, "      Nb Functions proved equivalent = %d\n",
NbEquiv);
    fprintf (stderr, "      Nb Functions proved non-equivalent = %d\n",
*NbErrors);
    fprintf (stderr, "      Nb Functions unproved = %d\n", 0);
    fprintf (stderr, "      MaxNb Nodes = %d\n", G_GnlMaxBddNode);
    fprintf (stderr, "      Nb CutVar = %d\n", G_NbCutVar);
    fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
    fprintf (stderr, "      Nb InvBridge Nodes = %d\n",
G_InvBridgeNodes);
    PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);
    return (GNL_OK);
}

fprintf (stderr, "\n");
fprintf (stderr, "      Nb Functions proved equivalent = %d\n",
NbEquiv);
fprintf (stderr, "      Nb Functions proved non-equivalent = %d\n",
    BListSize (OutErrors));
fprintf (stderr, "      Nb Functions unproved = %d\n", NbMayFalse);
fprintf (stderr, "      MaxNb Nodes = %d\n", G_GnlMaxBddNode);
fprintf (stderr, "      Nb CutVar = %d\n", G_NbCutVar);
fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
fprintf (stderr, "      Nb InvBridge Nodes = %d\n",
G_InvBridgeNodes);

#endif FND_2
    fprintf (stderr, "\n");
    fprintf (stderr, "      o Final False Negative Detection ... \n");
    if (GnlResolveFalseNegative (Gnl1, Gnl2, OutMayFalse, OutErrors,
ListBadPatterns))
        return (GNL_MEMORY_FULL);

NbMayFalse = BListSize (OutMayFalse);
NbEquiv = NbOut - NbMayFalse - BListSize (OutErrors);

```

gnlec.c

```
    fprintf (stderr, "\n");
    fprintf (stderr, "      Nb Functions proved equivalent = %d\n",
NbEquiv);
    fprintf (stderr, "      Nb Functions proved non-equivalent = %d\n",
      BListSize (OutErrors));
    fprintf (stderr, "      Nb Functions unproved = %d\n", BListSize
(OutMayFalse));
    fprintf (stderr, "      MaxNb Nodes = %d\n", G_GnlMaxBddNode);
    fprintf (stderr, "      Nb CutVar = %d\n", G_NbCutVar);
    fprintf (stderr, "      Nb Bridge Nodes = %d\n", G_BridgeNodes);
    fprintf (stderr, "      Nb InvBridge Nodes = %d\n", G_InvBridgeNodes);
#endif
}
#endif

*GnlErrors = OutErrors;
*GnlMayErrors = OutMayFalse;
*GnlBadPatterns = ListBadPatterns;

PrintResults (Gnl1, Gnl2, OutMayFalse, OutErrors, ListBadPatterns);

    return (GNL_OK);
}

/* ----- EOF ----- */
```

gnlec.h

```
/*-----*/
/*
 *      File:          gnlec.h
 *      Version:       1.1
 *      Documentation: -
 */
/*-----*/
#ifndef EC_H
#define EC_H

#define VALUE(v) ((v) == -1 ? 'u' : ((v) == 0 ? '0' : '1'))

#define GNL_NO_VAR      (char)0x00
#define GNL_VAR_GNL1    (char)0x0F
#define GNL_VAR_GNL2    (char)0xF0
#define GNL_VAR_BOTH    (char)0xFF

#define GNL_POLAR_NONE (char)0x00
#define GNL_POLAR_POS   (char)0x0F
#define GNL_POLAR_INV   (char)0xF0

typedef unsigned long Ulong64;
typedef Uint32 KEY_TYPE;

/*-----*/
/* Info stored on the Bdd node structure. */
/*-----*/
typedef struct BDD_REF_INFO_STRUCT
{
    GNL_NODE PosNode; /* Reference GNL_NODE of the Bdd for a
                        /* positive edge. */
    GNL_NODE NegNode; /* Reference GNL_NODE of the Bdd for a
                        /* negative edge. */
}
BDD_REF_INFO_REC, *BDD_REF_INFO;

#define GnlBddPosRefNode(b) (((BDD_REF_INFO)(b)->Hook)->PosNode)
#define SetGnlBddPosRefNode(b,l) (((BDD_REF_INFO)(b)->Hook)->PosNode = l)

#define GnlBddInvRefNode(b) (((BDD_REF_INFO)(b)->Hook)->NegNode)
#define SetGnlBddInvRefNode(b,l) (((BDD_REF_INFO)(b)->Hook)->NegNode = l)

/* ----- */
/* ADD_GNL_VAR_STRUCT */
/* ----- */
typedef struct ADD_GNL_VAR_STRUCT
{
    int        TagBdd;
    BDD        Bdd;           /* Bdd associated to this VAR */
    int        Height;
}
ADD_GNL_VAR_REC, *ADD_GNL_VAR;

#define GnlVarTagBdd(v)          (((((ADD_GNL_VAR)GnlVarHook (v))->TagBdd))
#define GnlVarBdd(v)             (((((ADD_GNL_VAR)GnlVarHook (v))->Bdd))
```

gnlec.h

```
#define GnlVarHeight (v)          (((((ADD_GNL_VAR)GnlVarHook (v))->Height))
#define GnlVarDepth (v)           (((((ADD_GNL_VAR)GnlVarHook (v))->Height))

#define SetGnlVarTagBdd(v,t)      (((((ADD_GNL_VAR)GnlVarHook (v))->TagBdd = t))
#define SetGnlVarBdd(v,b)         (((((ADD_GNL_VAR)GnlVarHook (v))->Bdd
= b))
#define SetGnlVarHeight(v,h)      (((((ADD_GNL_VAR)GnlVarHook (v))->Height = h))
#define SetGnlVarDepth(v,d)       (((((ADD_GNL_VAR)GnlVarHook (v))->Height = d))

/* -----
/* ADD_GNL_NODE_STRUCT
/* -----
typedef struct ADD_GNL_NODE_STRUCT
{
    GNL_VAR        NodesVar;
    int            TagBdd;
    BDD            Bdd;           /* Bdd associated to node's function */
    int            Height;
    BDD_VAR        BddVar;        /* Bdd var corresponding to a cut point */
    BLIST          Signatures;   /* results of random simulations */
    BLIST          EquivList;    /* list of potentially equivalent nodes */
    GNL_CONSTRAINT NegConstraint;
    GNL_CONSTRAINT PosConstraint;
    int            EquivTag;
    int            Value;
}
ADD_GNL_NODE_REC, *ADD_GNL_NODE;

#define GnlNodeVar(n)          (((((ADD_GNL_NODE)GnlNodeHook (n))->NodesVar))
#define GnlNodeTagBdd(n)        (((((ADD_GNL_NODE)GnlNodeHook (n))->TagBdd))
#define GnlNodeBdd(n)          (((((ADD_GNL_NODE)GnlNodeHook (n))->Bdd))
#define GnlNodeHeight(n)        (((((ADD_GNL_NODE)GnlNodeHook (n))->Height))
#define GnlNodeDepth(n)         (((((ADD_GNL_NODE)GnlNodeHook (n))->Height))
#define GnlNodeBreakBddVar(n)   (((((ADD_GNL_NODE)GnlNodeHook (n))->BddVar))
#define GnlNodeSignatures(n)    (((((ADD_GNL_NODE)GnlNodeHook (n))->Signatures))
#define GnlNodeEquivList(n)     (((((ADD_GNL_NODE)GnlNodeHook (n))->EquivList))
#define GnlNodeNegConstraint(n) (((((ADD_GNL_NODE)GnlNodeHook (n))->NegConstraint))
#define GnlNodePosConstraint(n) (((((ADD_GNL_NODE)GnlNodeHook (n))->PosConstraint))
#define GnlNodeEquivTag(n)      (((((ADD_GNL_NODE)GnlNodeHook (n))->EquivTag))
#define GnlNodeValue(n)         (((((ADD_GNL_NODE)GnlNodeHook (n))->Value))
```

```

#define SetGnlNodeVar(n,v) (((((ADD_GNL_NODE)GnlNodeHook (n))->NodesVar = v))
#define SetGnlNodeTagBdd(n,t) (((((ADD_GNL_NODE)GnlNodeHook (n))->TagBdd = t))
#define SetGnlNodeBdd(n,b) (((((ADD_GNL_NODE)GnlNodeHook (n))->Bdd = b))
#define SetGnlNodeHeight(n,h) (((((ADD_GNL_NODE)GnlNodeHook (n))->Height = h))
#define SetGnlNodeDepth(n,d) (((((ADD_GNL_NODE)GnlNodeHook (n))->Height = d))
#define SetGnlNodeBreakBddVar(n,b) (((((ADD_GNL_NODE)GnlNodeHook (n))->BddVar = b))
#define SetGnlNodeSignatures(n,s) (((((ADD_GNL_NODE)GnlNodeHook (n))->Signatures = s))
#define SetGnlNodeEquivList(n,l) (((((ADD_GNL_NODE)GnlNodeHook (n))->EquivList = l))
#define SetGnlNodeNegConstraint(n,c) (((((ADD_GNL_NODE)GnlNodeHook (n))->NegConstraint = c))
#define SetGnlNodePosConstraint(n,c) (((((ADD_GNL_NODE)GnlNodeHook (n))->PosConstraint = c))
#define IncrGnlNodeEquivTag(n) (((((ADD_GNL_NODE)GnlNodeHook (n))->EquivTag++))
#define SetGnlNodeEquivTag(n,t) (((((ADD_GNL_NODE)GnlNodeHook (n))->EquivTag = t))
#define SetGnlNodeValue(n,t) (((((ADD_GNL_NODE)GnlNodeHook (n))->Value = t))

/* -----
/* ADD_GNL_STRUCT
/* -----
typedef struct ADD_GNL_STRUCT
{
    int          TagBdd;
    GNL_VAR      ConstantVars[2];
    BLIST        RelevantInputs;
    BLIST        VirtualOutputs;
}
ADD_GNL_REC, *ADD_GNL;

#define GnlTagBdd(g) (((((ADD_GNL)GnlHook (g))->TagBdd)))
#define GnlConstant(g,i) (((((ADD_GNL)GnlHook (g))->ConstantVars[i]))
#define GnlRelevantInputs(g) (((((ADD_GNL)GnlHook (g))->RelevantInputs))
#define GnlVirtualOutputs(g) (((((ADD_GNL)GnlHook (g))->VirtualOutputs))

#define IncrGnlTagBdd(g) (((((ADD_GNL)GnlHook (g))->TagBdd++))
#define SetGnlTagBdd(g,t) (((((ADD_GNL)GnlHook (g))->TagBdd = t))
#define SetGnlConstant(g,i,v) (((((ADD_GNL)GnlHook (g))->ConstantVars[i] = v))
#define SetGnlRelevantInputs(g,i) (((((ADD_GNL)GnlHook (g))->RelevantInputs = i))
#define SetGnlVirtualOutputs(g,o) (((((ADD_GNL)GnlHook (g))->VirtualOutputs = o))

```

gnlec.h

```
/*-----*/
/* GNL_EC_CUT_POINT */
/*-----*/
typedef struct GNL_EC_CUT_POINT_STRUCT {
    BDD_VAR BddVar;           /* Bdd var corresponding to a cut point */
    GNL_NODE Node;            /* GNL_NODE corresponding to the bdd var*/
    BDD       BddFct;          /* Bdd corresponding to the function of
*/                                /* Node. */
    BDD       Support;
    int      NbNode;
}
GNL_EC_CUT_POINT_REC, *GNL_EC_CUT_POINT;

#define GnlEcCutPointBddVar(c)          ((c)->BddVar)
#define GnlEcCutPointNode(c)            ((c)->Node)
#define GnlEcCutPointBddFct(c)          ((c)->BddFct)
#define GnlEcCutPointSupport(c)         ((c)->Support)
#define GnlEcCutPointNbNode(c)          ((c)->NbNode)

#define SetGnlEcCutPointBddVar(c,v)   ((c)->BddVar = v)
#define SetGnlEcCutPointNode(c,v)     ((c)->Node = v)
#define SetGnlEcCutPointBddFct(c,v)   ((c)->BddFct = v)
#define SetGnlEcCutPointSupport(c,b)  ((c)->Support = b)
#define SetGnlEcCutPointNbNode(c,v)   ((c)->NbNode = v)

/* ----- EOF ----- */
#endif
```

gnlecmain.c

```
/*-----*/
/*
 *      File:          gnlecmain.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance: -
 */
/*
 */
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "blist.h"
#include "gnl.h"
#include "gnlopt.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnloption.h"

/*-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV      G_GnlEnv;

/*-----*/
/* GnlEcVerif */
/*-----*/
int GnlEcVerif ()
{
    int                  n;
    int                  i;
    BLIST    ListGnls1;
    BLIST    ListGnls2;
    GNL      Gnl1;
    GNL      Gnl2;
    BLIST    Outputs2;
    int                  NbMayErrors;
    BLIST    Gnl1MayErrors = NULL;
    BLIST    Gnl2MayErrors = NULL;
    int                  NbErrors;
    BLIST    Gnl1Errors = NULL;
    BLIST    Gnl2Errors = NULL;
    BLIST    GnlBadPatterns = NULL;
    int                  MaxNbNodes;
    void                  *Ptr;
    char                  *File1;
    char                  *File2;

    File2 = GnlEnvInput();
```

gnlecmain.c

```
File1 = GnlEnvReference();

fprintf (stderr, "\n# File = %s\n", File1);
fprintf (stderr, "\n# Analyzing [%s] ... \n", File1);
if (GnlRead (File1, &ListGnls1))
{
    fprintf (stderr, "# ERROR: Verification aborted.\n");
    exit (1);
}

fprintf (stderr, "\n# Analyzing [%s] ... \n", File2);
if (GnlRead (File2, &ListGnls2))
{
    fprintf (stderr, "# ERROR: Verification aborted.\n");
    exit (1);
}

if (BListSize (ListGnls1) != BListSize (ListGnls2))
{
    fprintf (stderr,
            "# ERROR: netlists have different number of modules\n");
    return (1);
}

MaxNbNodes = GnlEnvMaxBddNode();
if (MaxNbNodes == 0)
    MaxNbNodes = 500;

for (n=0; n<BListSize (ListGnls1); n++)
{
    Gnl1 = (GNL)BListElt (ListGnls1, n);
    Gnl2 = (GNL)BListElt (ListGnls2, n);

    if (GnlNbIn (Gnl1) != GnlNbIn (Gnl2))
    {
        fprintf (stderr,
                "# ERROR: netlists have different number of inputs\n");
        return (1);
    }

    if (GnlNbOut (Gnl1) != GnlNbOut (Gnl2))
    {
        fprintf (stderr,
                "# ERROR: netlists have different number of outputs\n");
        return (1);
    }

    fprintf (stderr, "\n# Verifying Logic [%s] Vs. [%s]..\n",
            GnlName (Gnl1), GnlName (Gnl2));
    if (GnlEquivCheckGnl (Gnl1, Gnl2, MaxNbNodes,
                          &NbMayErrors, &Gnl1MayErrors, &Gnl2MayErrors,
                          &NbErrors, &Gnl1Errors, &Gnl2Errors, &GnlBadPatterns))
    {
        fprintf (stderr, "# ERROR: Critical Problem During Verification\n");
        exit (1);
    }
}
```

gnlecmain.c

```
    }  
  
    return (0);  
}
```

© 2000-2004, Texas Instruments Incorporated - All Rights Reserved

gnlerror.c

```
/*-----*/  
/*  
/*      File:          gnlerror.c  
/*      Modifications: -  
/*      Documentation: -  
/*  
/*      Description:  
/*-----*/  
  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
  
/*-----*/  
/* GnlError  
/*-----*/  
void GnlError (Id)  
    int      Id;  
{  
    fprintf (stderr, "\n# GNL-ERROR<%d>\n", Id);  
    exit (1);  
}
```

gnlestim.c

```
/*-----*/
/*
 *      File:          gnlestim.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description: *
 */
/*-----*/
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "gnloption.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlestim.h"
#include "gnlmap.h"
#include "time.h"

#include "blist.e"

#include "libutil.e"
#include "timecomp.e"

/*-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV          G_GnlEnv;
extern BDD_PTR           GetBddPtrFromBdd ();

/*-----*/
/* FORWARD */
/*-----*/
GNL_STATUS   GnlBestAreaMapNode ();
GNL_STATUS   GnlBestDepthMapNode ();
void         GnlAddPinCapacitance ();
GNL_STATUS   GnlGetMaxMinDelaysFromNode ();
void         GnlGetMaxTechnologyDepthFromNode ();

/*-----*/
/* GnlCreatePathComponent */
/*-----*/
GNL_STATUS GnlCreatePathComponent (NewPathCompo)
    GNL_PATH_COMPONENT *NewPathCompo;
{
```

```

if ((*NewPathCompo = (GNL_PATH_COMPONENT)
     calloc (1, sizeof(GNL_PATH_COMPONENT_REC))) == NULL)
    return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlCreateAssocTraversalInfo
*/
-----*/
GNL_STATUS GnlCreateAssocTraversalInfo (NewTraversalInfo)
    GNL_ASSOC_TRAVERSAL_INFO *NewTraversalInfo;
{
    if ((*NewTraversalInfo = (GNL_ASSOC_TRAVERSAL_INFO)
         calloc (1, sizeof(GNL_ASSOC_TRAVERSAL_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    (*NewTraversalInfo) ->Component = (GNL_COMPONENT) NULL;
    (*NewTraversalInfo) ->Hook = NULL;

    return (GNL_OK);
}

/*-----
/* GnlFreeAssocTraversalInfo
*/
-----*/
void GnlFreeAssocTraversalInfo (AssocTraversalInfo, HookDelete)
    GNL_ASSOC_TRAVERSAL_INFO *AssocTraversalInfo;
    VoidFctPtr             HookDelete;
{
    if (!(*AssocTraversalInfo))
        return;

    if ((*AssocTraversalInfo) ->Hook)
        HookDelete (&(*AssocTraversalInfo) ->Hook));

    free ((char *)*AssocTraversalInfo);

    (*AssocTraversalInfo) = NULL;
}

/*-----
/* GnlCreateVarTraversalInfo
*/
-----*/
GNL_STATUS GnlCreateVarTraversalInfo (NewTraversalInfo)
    GNL_VAR_TRAVERSAL_INFO *NewTraversalInfo;
{
    if ((*NewTraversalInfo = (GNL_VAR_TRAVERSAL_INFO)
         calloc (1, sizeof(GNL_VAR_TRAVERSAL_INFO_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

```

gnlestim.c

```
}

/*-----*/
/* GnlFreeVarTraversalInfo */
/*-----*/
void GnlFreeVarTraversalInfo (VarTraversalInfo, HookDelete)
    GNL_VAR_TRAVERSAL_INFO      *VarTraversalInfo;
    VoidFctPtr                  HookDelete;
{

    if (!(*VarTraversalInfo))
        return;

    if ((*VarTraversalInfo)->ListAssoc)
        BListQuickDelete (&(*VarTraversalInfo)->ListAssoc);

    if ((*VarTraversalInfo)->ListLeftAssign)
        BListQuickDelete (&(*VarTraversalInfo)->ListLeftAssign);

    if ((*VarTraversalInfo)->Hook)
        HookDelete (&(*VarTraversalInfo)->Hook);

    free ((char *) *VarTraversalInfo);

    (*VarTraversalInfo) = NULL;
}

/*-----*/
/* GnlVarTraversalInfoAddAssoc */
/*-----*/
GNL_STATUS GnlVarTraversalInfoAddAssoc (Var, Assoc)
    GNL_VAR Var;
    GNL_ASSOC Assoc;
{
    BLIST NewList;

    if (!Var)
        return (GNL_OK);

    if (!GnlVarTraversalInfoListAssoc (Var))
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        SetGnlVarTraversalInfoListAssoc (Var, NewList);
    }

    if (BListAddElt (GnlVarTraversalInfoListAssoc (Var), (int) Assoc))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----*/
/* GnlVarTraversalInfoAddLeftVarAssign */
/*-----*/
```

gnlestim.c

```
GNL_STATUS GnlVarTraversalInfoAddLeftVarAssign (Var, LeftAssigned)
    GNL_VAR Var;
    GNL_VAR LeftAssigned;
{
    BLIST NewList;

    if (!GnlVarTraversalInfoListLeftAssign (Var))
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        SetGnlVarTraversalInfoListLeftAssign (Var, NewList);
    }

    if (BListAddElt (GnlVarTraversalInfoListLeftAssign (Var),
                     (int)LeftAssigned))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlGetGnlMapAreaFromNode
/*-----*/
double GnlGetGnlMapAreaFromNode (Gnl, OutVar, Node)
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_NODE   Node;
{
    int           i;
    BDD          Bdd;
    BDD_PTR      BddPtr;
    BLIST        ListDeriveCells;
    float        BestArea;
    GNL_NODE     Son;
    GNL_VAR      Var;
    LIB_DERIVE_CELL BestCell;
    int          Depth;

    switch (GnlNodeOp (Node)) {
        case GNL_VARIABLE:
        case GNL_CONSTANTE:
        case GNL_WIRE:
            return (0.0);

        case GNL_NOT:
        case GNL_AND:
        case GNL_OR:
        case GNL_NOR:
        case GNL_NAND:
        case GNL_XOR:
        case GNL_XNOR:
            if (GnlNodeHook (Node) && GnlMapNodeInfoCutVar (Node))
            {
                Var = GnlMapNodeInfoCutVar (Node);
```

```

        if (Var != OutVar)
            {
                return (0.0);
            }
        Bdd = GnlMapNodeInfoBestBdd (Node);
        if (!Bdd)
            return (0.0);
        BddPtr = GetBddPtrFromBdd (Bdd);
        if ((Bdd == bdd_zero ()) ||
            (Bdd == bdd_one ()))
            return (0.0);
        ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
        GnlGetBestAreaCell (ListDeriveCells, Bdd, &BestCell,
                            &BestArea, &Depth);
        return ((double)BestArea);
    }
    fprintf (stderr, " ERROR: in GnlGetGnlMapAreaFromNode");
    return;
}

default:
    GnlError (9 /* Unknown node */);
    return;
}

/*-----*/
/* GnlGetComponentArea */
/*-----*/
double GnlGetComponentArea (Component)
    GNL_COMPONENT          Component;
{
    LIBC_CELL              Cell;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_BUF_COMPONENT      BufCompo;
    double                 CellArea;

    switch (GnlComponentType (Component)) {
        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT)Component;
            Cell = GnlSeqCompoInfoCell (SeqCompo);
            CellArea = (double)LibCellArea (Cell);
            return (CellArea);

        case GNL_USER_COMPO:
            return (0.0);

        case GNL_TRISTATE_COMPO:
            TriStateCompo = (GNL_TRISTATE_COMPONENT)Component;
            Cell = GnlTriStateCompoInfoCell (TriStateCompo);
            CellArea = (double)LibCellArea (Cell);
            return (CellArea);

        case GNL_BUF_COMPO:
            BufCompo = (GNL_BUF_COMPONENT)Component;
            Cell = GnlBufCompoInfoCell (BufCompo);

```

```

        CellArea = (double)LibCellArea (Cell);
        return (CellArea);

    case GNL_MACRO_COMPO:
        return (0.0);

    default:
        GnlError (12 /* Unknown component */);
    }
}

<賓客
/*-----*/
/* GnlGetGnlMapArea
 *-----*/
/* This procedure computes the technology area of the module 'Gnl' and */
/* returns a double value corresponding to it. It simply adds the */
/* cells area created during the mapping phase. */
/*-----*/
double GnlGetGnlMapArea (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_VAR       VarI;
    GNL_FUNCTION  FunctionI;
    double        Area;
    GNL_COMPONENT ComponentI;

    Area = 0.0;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        Area += GnlGetGnlMapAreaFromNode (Gnl, VarI,
                                         GnlFunctionOnSet (FunctionI));
    }

    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);
        Area += GnlGetComponentArea (ComponentI);
    }

    return (Area);
}

<賓客
/*-----*/
/* GnlGetGnlMapAreaInGnl
 *-----*/
void GnlGetGnlMapAreaInGnl (Gnl)
    GNL          Gnl;
{
    int           i;
    double        GnlArea;
    GNL_COMPONENT ComponentI;
}

```

gnlestim.c

```
GNL_USER_COMPONENT           UserCompo;
GNL_SEQUENTIAL_COMPONENT    SeqCompo;
GNL_TRISTATE_COMPONENT      TristateCompo;
GNL_BUF_COMPONENT            BufCompo;
LIBC_CELL                    Cell;
BLIST                        Components;

GnlArea = 0.0;

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    switch (GnlComponentType (ComponentI)) {
        case GNL_USER_COMPO:
            UserCompo = (GNL_USER_COMPONENT)ComponentI;
            /* If not a black box or cell we do take it into */
            /* account. */
            Cell = GnlUserComponentCellDef (UserCompo);
            if (!Cell)
                continue;
            break;

        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
            Cell = GnlSeqCompoInfoCell (SeqCompo);
            break;

        case GNL_TRISTATE_COMPO:
            TristateCompo = (GNL_TRISTATE_COMPONENT)ComponentI;
            Cell = GnlTriStateCompoInfoCell (TristateCompo);
            break;

        case GNL_BUF_COMPO:
            BufCompo = (GNL_BUF_COMPONENT)ComponentI;
            Cell = GnlBufCompoInfoCell (BufCompo);
            break;
    }

    if (!Cell)
        continue;

    GnlArea += (double)LibCellArea (Cell);
}

SetGnlArea (Gnl, GnlArea);

GnlComputeMaxFanoutInGnl (Gnl);

}

/*-----*/
```

gnlestim.c

```
/* GnlGetGnlMapAreaInNwRec */  
/*-----*/  
/* This procedure computes the technology area of the module 'Gnl' and */  
/* returns a double value corresponding to it. It simply adds the */  
/* cells area created during the mapping phase. */  
/*-----*/  
void GnlGetGnlMapAreaInNwRec (Nw, Gnl)  
    GNL_NETWORK Nw;  
    GNL Gnl;  
{  
    int i;  
    GNL TopGnl;  
    GNL_COMPONENT ComponentI;  
    GNL_USER_COMPONENT UserCompoI;  
    GNL GnlCompoI;  
    BLIST Components;  
  
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))  
        return ;  
  
    SetGnlTag (Gnl, GnlNetworkTag (Nw));  
  
    GnlGetGnlMapAreaInGnl (Gnl);  
  
    Components = GnlComponents (Gnl);  
  
    for (i=0; i<BListSize (Components); i++)  
    {  
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);  
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)  
            continue;  
        UserCompoI = (GNL_USER_COMPONENT)ComponentI;  
        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);  
  
        if (!GnlCompoI)  
            continue;  
  
        GnlGetGnlMapAreaInNwRec (Nw, GnlCompoI);  
    }  
}  
/*-----*/  
/* GnlGetGnlMapAreaInNw */  
/*-----*/  
/* This procedure computes the technology area of the module 'Gnl' and */  
/* returns a double value corresponding to it. It simply adds the */  
/* cells area created during the mapping phase. */  
/*-----*/  
void GnlGetGnlMapAreaInNw (Nw)  
    GNL_NETWORK Nw;  
{  
    GNL TopGnl;  
  
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
```

gnlestim.c

```
TopGnl = GnlNetworkTopGnl (Nw) ;

    GnlGetGnlMapAreaInNwRec (Nw, TopGnl) ;
}

/*-----*/
/* GnlGetMaxTechnologyDepthFromNodeVar */
/*-----*/
void GnlGetMaxTechnologyDepthFromNodeVar (Gnl, OutVar, Node)
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_NODE   Node;
{
    GNL_VAR      Var;
    GNL_FUNCTION FunctionI;
    GNL_NODE   NodeI;

    Var = (GNL_VAR) GnlNodeSons (Node);

    if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
    {
        SetGnlMapNodeInfoDepth (Node, 0);
        return;
    }

    if (GnlVarFunction (Var) == NULL)
    {
        SetGnlMapNodeInfoDepth (Node, 0);
        return;
    }

    FunctionI = GnlVarFunction (Var);
    NodeI = GnlFunctionOnSet (FunctionI);
    GnlGetMaxTechnologyDepthFromNode (Gnl, Var, NodeI);
    SetGnlMapNodeInfoDepth (Node, GnlMapNodeInfoDepth (NodeI));
}

/*-----*/
/* GnlGetGnlMaxDepthFromNodeOp */
/*-----*/
void GnlGetMaxTechnologyDepthFromNodeOp (Gnl, OutVar, Node)
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_NODE   Node;
{
    int           i;
    int           j;
    int           MaxDepth;
    GNL_NODE   NodeJ;
    GNL_VAR     VarI;
    GNL_VAR     VarCellI;
    BLIST       CellInputs;
    BLIST       LibVarSupport;
    BLIST       NodeSupport;
    BDD         Bdd;
```

gnlestim.c

```
BDD_PTR          BddPtr;
BLIST            ListDeriveCells;
float            BestArea;
GNL_NODE         Son;
GNL_VAR          Var;
LIB_DERIVE_CELL BestCell;
GNL_FUNCTION     FunctionI;
GNL_NODE         NodeI;
int              Depth;

if (!GnlNodeHook (Node) || !GnlMapNodeInfoCutVar (Node))
{
    fprintf (stderr, " ERROR: in GnlGetMaxTechnologyDepthFromNodeOp");
}

Var = GnlMapNodeInfoCutVar (Node);
if (Var != OutVar)
{
    if (GnlVarFunction (Var) == NULL)
    {
        SetGnlMapNodeInfoDepth (Node, 0);
        return ;
    }
    FunctionI = GnlVarFunction (Var);
    NodeI = GnlFunctionOnSet (FunctionI);
    GnlGetMaxTechnologyDepthFromNode (Gnl, Var, NodeI);
    return;
}

Bdd = GnlMapNodeInfoBestBdd (Node);
if (!Bdd)
    return;
BddPtr = GetBddPtrFromBdd (Bdd);
if ((Bdd == bdd_zero ()) || (Bdd == bdd_one ()))
{
    SetGnlMapNodeInfoDepth (Node, 0);
    return ;
}

ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
GnlGetBestAreaCell (ListDeriveCells, Bdd, &BestCell, &BestArea,
&Depth);

MaxDepth = 0;

NodeSupport = GnlMapNodeInfoNodeSupport (Node);
LibVarSupport = GnlMapNodeInfoLibVarSupport (Node);
CellInputs = LibDeriveCellInputs (BestCell);
for (i=0; i<BListSize (CellInputs); i++)
{
    VarI = (GNL_VAR)BListElt (CellInputs, i);
    for (j=0; j<BListSize (LibVarSupport); j++)
    {
        VarCellI = (GNL_VAR)BListElt (LibVarSupport, j);
        if (VarI == VarCellI)
            break;
    }
}
```

```

        }
        NodeJ = (GNL_NODE)BListElt (NodeSupport, j);
        if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
            VarI = (GNL_VAR)GnlNodeSons (NodeJ);
        else
            VarI = GnlMapNodeInfoCutVar (NodeJ);
        GnlGetMaxTechnologyDepthFromNode (Gnl, VarI, NodeJ);
        if (GnlMapNodeInfoDepth (NodeJ) > MaxDepth)
            MaxDepth = GnlMapNodeInfoDepth (NodeJ);
    }

    MaxDepth++;
    SetGnlMapNodeInfoDepth (Node, MaxDepth);
}

/*-----*/
/* GnlGetGnlMaxDepthFromNode */
/*-----*/
void GnlGetMaxTechnologyDepthFromNode (Gnl, OutVar, Node)
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_NODE   Node;
{
    /* If previously computed then we return. */
    if (GnlMapNodeInfoDepth (Node) != 0)
        return;

    switch (GnlNodeOp (Node)) {
        case GNL_VARIABLE:
            GnlGetMaxTechnologyDepthFromNodeVar (Gnl, OutVar, Node);
            return;

        case GNL_CONSTANTE:
        case GNL_WIRE:
            SetGnlMapNodeInfoDepth (Node, 0);
            return;

        case GNL_NOT:
        case GNL_AND:
        case GNL_OR:
        case GNL_NOR:
        case GNL_NAND:
        case GNL_XOR:
        case GNL_XNOR:
            GnlGetMaxTechnologyDepthFromNodeOp (Gnl, OutVar, Node);
            return;

        default:
            GnlError (9 /* Unknown node */);
            return;
    }
}

/*-----*/
/* GnlGetMaxTechnologyDepth */
/*-----*/

```

gnlestim.c

```
/* This procedure computes the maximum depth between cells which have */
/* been created during the technology mapping phase. */
/*-----*/
int GnlGetMaxTechnologyDepth (Gnl)
    GNL          Gnl;
{
    int          i;
    GNL_VAR      VarI;
    GNL_FUNCTION FunctionI;
    int          MaxDepth;
    GNL_NODE     NodeI;

    MaxDepth = 0;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR) BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        NodeI = GnlFunctionOnSet (FunctionI);
        GnlGetMaxTechnologyDepthFromNode (Gnl, VarI, NodeI);
        if (GnlMapNodeInfoDepth (NodeI) > MaxDepth)
        {
            MaxDepth = GnlMapNodeInfoDepth (NodeI);
        }
    }

    return (MaxDepth);
}

/*-----*/
/* GnlComputeGnlGateArea */
/*-----*/
/* Computes the active area of the GNL 'Gnl'. The active area is */
/* composed of the cells area (Dffs, Latches, Tristates, Buffers). */
/*-----*/
double GnlComputeGnlGateArea (Gnl)
    GNL          Gnl;
{
    int          i;
    double       Area;
    GNL_COMPONENT Component;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_USER_COMPONENT UserCompo;
    GNL           GnlCompo;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_BUF_COMPONENT BufCompo;
    LIBC_CELL     Cell;

    Area = 0.0;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        Component = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
```

gnlestim.c

```
switch (GnlComponentType (Component)) {
    case GNL_SEQUENTIAL_COMPO:
        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) Component;
        Cell = GnlSeqCompoInfoCell (SeqCompo);
        Area += (double)LibCellArea (Cell);
        break;

    case GNL_TRISTATE_COMPO:
        TriStateCompo = (GNL_TRISTATE_COMPONENT) Component;
        Cell = GnlTriStateCompoInfoCell (TriStateCompo);
        Area += (double)LibCellArea (Cell);
        break;

    case GNL_BUF_COMPO:
        BufCompo = (GNL_BUF_COMPONENT) Component;
        Cell = GnlBufCompoInfoCell (BufCompo);
        Area += (double)LibCellArea (Cell);
        break;

    case GNL_USER_COMPO:
        UserCompo = (GNL_USER_COMPONENT) Component;
        GnlCompo = GnlUserComponentGnlDef (UserCompo);
        if (GnlCompo)
            continue;

        Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
        if (Cell)
        {
            Area += (double)LibCellArea (Cell);
        }
        break;

    default:
        break;
}
}

return (Area);
}

/*-----
 * GnlGetNetworkAreaRec
 *-----
double GnlGetNetworkGateAreaRec (Gnl)
    GNL Gnl;
{
    double          Area;
    GNL             TopGnl;
    BLIST           Components;
    GNL_COMPONENT   ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    int             i;
    GNL             GnlCompoI;

    Area = GnlComputeGnlGateArea (Gnl);
```

```

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (!GnlCompoI)
        continue;

    Area += GnlGetNetworkGateAreaRec (GnlCompoI);
}

return (Area);
}

/*-----
/* GnlGetNetworkGateArea
/*-----
/* This procedure computes the area estimation related to the global
/* netlist 'GlobalNetwork' using library cells of 'GnlLibc'.
/*-----*/
double GnlGetNetworkGateArea (Nw)
    GNL_NETWORK Nw;
{
    double      Area;
    GNL         TopGnl;

    TopGnl = GnlNetworkTopGnl (Nw);

    Area = GnlGetNetworkGateAreaRec (TopGnl);

    return (Area);
}

/*-----
/* GnlCreateLibraryHashTable
/*-----
/* This procedure scans the LIBC 'GnlLibc' and stores each cell in a
/* Hash table. Each cell is stored according to its name.
/*-----*/
#define LIB_HASH_TABLE_SIZE 100
GNL_STATUS GnlCreateLibraryHashTable (GnlLibc, LibHashTable)
    LIBC_LIB     GnlLibc;
    BLIST       *LibHashTable;
{
    int          i;
    LIBC_CELL    CellI;
    unsigned int Key;
    BLIST       NewList;

```

```

if (BListCreateWithSize (LIB_HASH_TABLE_SIZE, LibHashTable))
    return (GNL_MEMORY_FULL);

for (i=0; i<LIB_HASH_TABLE_SIZE; i++)
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (*LibHashTable, (int)NewList))
        return (GNL_MEMORY_FULL);
}

CellI = LibCells (GnlLibc);
for (; CellI != NULL; CellI = LibCellNext (CellI))
{
    Key = KeyOfName (LibCellName (CellI), LIB_HASH_TABLE_SIZE);
    NewList = (BLIST)BListElt (*LibHashTable, Key);
    if (BListAddElt (NewList, (int)CellI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlUpdateVarAssocList
 */
/*
 * This procedure adds the association 'Assoc' in the list of assoc of
 * the actual 'Actual'. If the actual is a GNL_VAR (a simple signal)
 * then we add it simply in 'GnlVarTraversalInfoListAssoc (Actual)' else*
 * the actual is a GNL_NODE corresponding to a concatenation of GNL_VAR.*/
/* Then for each son GNL_VAR we add the association in their list assoc */
/* field. */
/*
 * GNL_STATUS GnlUpdateVarAssocList (Actual, Assoc)
 * GNL_VAR Actual;
 * GNL_ASSOC Assoc;
{
    int i;
    GNL_VAR SplitActualI;
    BLIST ListSplitActuals;

    if (!Actual)
        return (GNL_OK);

    /* If the var is VSS or VDD it is not useful to store the places
     * where it is used. */
    if (GnlVarIsVar (Actual))
        if (GnlVarIsVss (Actual) || GnlVarIsVdd (Actual))
            return (GNL_OK);

    if (GnlVarIsVar (Actual))
        if (GnlVarIsVss (Actual) || GnlVarIsVdd (Actual))
            return (GNL_OK);
    /* If 'Actual' is a GNL_VAR and not a GNL_NODE (GNL_CONCAT) */

```

gnlestim.c

```
if (GnlVarIsVar (Actual))
{
    if (GnlVarTraversalInfoAddAssoc (Actual, Assoc))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

ListSplitActuals = GnlNodeSons ((GNL_NODE)Actual);
for (i=0; i<BListSize (ListSplitActuals); i++)
{
    SplitActualI = (GNL_VAR)BListElt (ListSplitActuals, i);
    if (GnlVarTraversalInfoAddAssoc (SplitActualI, Assoc))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlAddAssocTraversalInfo
*/
/*-----
/* This procedure scans all the GNL_ASSOC objects of the interface and */
/* creates a Timing Info structure and stores it on the hook field of */
/* the association. */
/* Moreover, for each actual var we add its corresponding GNL_ASSOC in */
/* its field 'GnlVarTraversalInfoListAssoc'. */
*/
GNL_STATUS GnlAddAssocTraversalInfo (Compo, Interface)
    GNL_COMPONENT Compo;
    BLIST Interface;
{
    int i;
    GNL_ASSOC AssocI;
    GNL_ASSOC_TRAVERSAL_INFO NewTraversalInfo;
    GNL_VAR Actual;

    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);

        if (!AssocI)
            continue;
        Actual = GnlAssocActualPort (AssocI);
        if (!Actual)
            continue;

        /*
        if (GnlVarIsVar (Actual))
            if (GnlVarIsVss (Actual) || GnlVarIsVdd (Actual))
                continue;
        */

        if (GnlCreateAssocTraversalInfo (&NewTraversalInfo))
            return (GNL_MEMORY_FULL);

        /* we hook the field with this new structure. */
    }
}
```

gnlestim.c

```
        SetGnlAssocHook (AssocI, NewTraversalInfo);

        SetGnlAssocTraversalInfoComponent (AssocI, Compo);

        if (GnlUpdateVarAssocList (Actual, AssocI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlAddVarTraversalInfo
*/
/*-----*/
/* This procedure creates and adds traversal info structure for each object*/
/* GNL_VAR of the current 'Gnl' on the Hook field of the vars. */
/*-----*/
GNL_STATUS GnlAddVarTraversalInfo (Gnl)
    GNL          Gnl;
{
    BLIST      HashTableNames;
    int         i;
    BLIST      BucketI;
    int         j;
    GNL_VAR     VarJ;
    GNL_VAR_TRAVERSAL_INFO   NewTraversalInfo;

    HashTableNames = GnlHashNames (Gnl);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST)BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);
            if (GnlCreateVarTraversalInfo (&NewTraversalInfo))
                return (GNL_MEMORY_FULL);

            /* we hook the field with this new structure.
*/
            SetGnlVarHook (VarJ, NewTraversalInfo);
        }
    }

    return (GNL_OK);
}

/*-----
/* GnlUpdateListLeftAssignToVarInGnl
*/
/*-----*/
/* This procedure updates the list of LeftVarAssign for each GNL_VAR */
/* 'Var' in any assignement. */
/* Ex: 'assign O1 = Var' */
/*      'assign O2 = Var' */
/*      'assign O3 = Var' */
/* The list associated to 'Var' will be {'o1', 'o2', 'o3'}. This is */

```

gnlestim.c

```
/* usefull for bottom-up traversal. */  
/*-----*/  
GNL_STATUS GnlUpdateListLeftAssignToVarInGnl (Gnl)  
    GNL          Gnl;  
{  
    BLIST        HashTableNames;  
    int          i;  
    BLIST        BucketI;  
    int          j;  
    GNL_VAR      VarI;  
    GNL_NODE     FunctionNode;  
    GNL_VAR      RightVar;  
    GNL_FUNCTION Function;  
  
    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)  
    {  
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  
        Function = GnlVarFunction (VarI);  
        if (!Function)  
            continue;  
  
        FunctionNode = GnlFunctionOnSet (Function);  
  
        if (GnlNodeOp (FunctionNode) == GNL_CONSTANTE)  
            continue;  
  
        if (GnlNodeOp (FunctionNode) != GNL_VARIABLE)  
        {  
            fprintf (stderr,  
                    " ERROR1: non simple Boolean equation detected\n");  
            exit (1);  
        }  
  
        RightVar = (GNL_VAR)GnlNodeSons (FunctionNode);  
  
        if (GnlVarTraversalInfoAddLeftVarAssign (RightVar, VarI))  
            return (GNL_MEMORY_FULL);  
    }  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGetInterfaceFromGnlCompo */  
/*-----*/  
GNL_STATUS GnlGetInterfaceFromGnlCompo (GNL_COMPONENT Component,  
                                         BLIST      *Interface)  
{  
    GNL_USER_COMPONENT      UserCompo;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    GNL_TRISTATE_COMPONENT  TriStateCompo;  
    GNL_BUF_COMPONENT        BufCompo;  
  
    switch (GnlComponentType (Component)) {
```

```

    case GNL_SEQUENTIAL_COMPO:
        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) Component;
        *Interface = GnlSequentialCompoInterface (SeqCompo);
        break;

    case GNL_USER_COMPO:
        UserCompo = (GNL_USER_COMPONENT) Component;
        *Interface = GnlUserComponentInterface (UserCompo);
        break;

    case GNL_TRISTATE_COMPO:
        TriStateCompo = (GNL_TRISTATE_COMPONENT) Component;
        *Interface = GnlTriStateInterface (TriStateCompo);
        break;

    case GNL_BUF_COMPO:
        BufCompo = (GNL_BUF_COMPONENT) Component;
        *Interface = GnlBufInterface (BufCompo);
        break;

    case GNL_MACRO_COMPO:
        *Interface = NULL;
        break;

    default:
        *Interface = NULL;
        GnlError (12 /* Unknown component */);
        break;
    }

    return (GNL_OK);
}

/*-----
/* GnlGetCellFromGnlCompo
/*-----*/
GNL_STATUS GnlGetCellFromGnlCompo (GNL_COMPONENT          Component,
                                    LIBC_CELL      *Cell)
{
    GNL_USER_COMPONENT      UserCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_BUF_COMPONENT       BufCompo;

    switch (GnlComponentType (Component)) {

        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT) Component;
            *Cell = GnlSeqCompoInfoCell (SeqCompo);
            break;

        case GNL_USER_COMPO:
            UserCompo = (GNL_USER_COMPONENT) Component;
            *Cell = GnlUserComponentCellDef (UserCompo);
            break;
    }
}

```

```

    case GNL_TRISTATE_COMPO:
        TriStateCompo = (GNL_TRISTATE_COMPONENT) Component;
        *Cell = GnlTriStateCompoInfoCell (TriStateCompo);
        break;

    case GNL_BUF_COMPO:
        BufCompo = (GNL_BUF_COMPONENT) Component;
        *Cell = GnlBufCompoInfoCell (BufCompo);
        break;

    case GNL_MACRO_COMPO:
        *Cell = NULL;
        break;

    default:
        *Cell = NULL;
        GnlError (12 /* Unknown component */);
        break;
    }

    return (GNL_OK);
}

/*-----
/* GnlAddNetlistInfoForTraversalInComponentRec
/*-----
/* This procedure analyzes recursively the netlist 'Gnl' and for each */
/* GNL_VAR we create its timing info structure and update its field */
/* 'GnlVarTraversalInfoListAssoc' which is the list of the GNL_ASSOC */
/* where this var appears in the Actual form.
/*-----*/
GNL_STATUS GnlAddNetlistInfoForTraversalInComponentRec (Nw, Gnl)
    GNL_NETWORK      Nw;
    GNL             Gnl;
{
    GNL             TopGnl;
    BLIST           Components;
    GNL_COMPONENT   ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    int              i;
    int              j;
    GNL             GnlCompoI;
    unsigned int     Key;
    BLIST           NewList;
    LIBC_CELL       CellJ;
    BLIST           Interface;

    /* Already traversed and considered this 'Gnl'.
     * if (GnlTag (Gnl) == GnlNetworkTag (Nw))
     *     return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));
    /* we add timing info on each var of the 'Gnl'.
     */
}

```

```

if (GnlAddVarTraversalInfo (Gnl))
    return (GNL_MEMORY_FULL);

/* Updating left assign var list for each Var. */
if (GnlUpdateListLeftAssignToVarInGnl (Gnl))
    return (GNL_MEMORY_FULL);

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    /* We create timing info structures for each assoc of the      */
    /* component interface.                                         */
    GnlGetInterfaceFromGnlCompo (ComponentI, &Interface);
    if (GnlAddAssocTraversalInfo (ComponentI, Interface))
        return (GNL_MEMORY_FULL);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    /* If the component 'UserCompoI' has no Gnl definition.      */
    if (!GnlCompoI)
    {
        if (!GnlUserComponentCellDef (UserCompoI))
        {
            fprintf (stderr,
" WARNING: component <%s> has no corresponding library cell\n",
                    GnlUserComponentName (UserCompoI));
        }
        continue;
    }

    /* It is a user box and we scan recursively.                  */
    if (GnlAddNetlistInfoForTraversalInComponentRec (Nw, GnlCompoI))
        return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 * GnlAddNetlistInfoForTraversalInComponent
 */
/*
 * This procedure analyzes recursively the netlist 'Gnl' and for each   */
/* GNL_VAR we create its timing info structure and update its field   */
/* 'GnlVarTraversalInfoListAssoc' which is the list of the GNL_ASSOC   */
/* where this var appears in the Actual form.                         */
/*
 * GNL_STATUS GnlAddNetlistInfoForTraversalInComponent (Nw, Gnl, GnlLibc)
 * GNL_NETWORK      Nw;
 * GNL             Gnl;
 * LIBC_LIB        GnlLibc;
{

```

gnlestim.c

```
int i;
BLIST NewList;

/* Resetting the Tag for recursive traversal.
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

if (GnlAddNetlistInfoForTraversalInComponentRec (Nw, Gnl))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*#ifdef TIMING_ESTIMATION*/
/*-----*/
/* GnlGetSourcesOfVar */
/*-----*/
GNL_STATUS GnlGetSourcesOfVar (Var, AssocSources, RightVarAssigneds)
    GNL_VAR Var;
    BLIST *AssocSources;
    BLIST *RightVarAssigneds;
{
    int i;
    GNL_ASSOC AssocI;
    LIBC_CELL Cell;
    GNL_COMPONENT Compo;
    GNL_VAR Formal, VarI;
    GNL_VAR TargetVar;
    LIBC_PIN Pin;
    BLIST ListAssoc;
    GNL_FUNCTION Function;
    GNL_NODE FunctionNode;
    GNL_STATUS GnlStatus;

    if (!Var)
        return (GNL_OK);

    ListAssoc = GnlVarTraversalInfoListAssoc (Var);

    if (BListCreateWithSize (1, AssocSources))
        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (1, RightVarAssigneds))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListAssoc); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (ListAssoc, i);
        VarI = GnlAssocActualPort (AssocI);
        if (GnlVarIsVar (VarI))
            if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
                continue;
        Compo = GnlAssocTraversalInfoComponent (AssocI);

        Formal = GnlAssocFormalPort (AssocI);
        if (GnlStatus = GnlGetCellFromGnlCompo (Compo, &Cell))


```

gnlestim.c

```
        return (GnlStatus);

    if (GnlComponentType (Compo) == GNL_USER_COMPO &&
        !GnlUserComponentGnlDef ((GNL_USER_COMPONENT) Compo) &&
        !Cell)
    {
        fprintf (stderr,
            " WARNING: black box <%s %s> may modify final estimation\n",
            GnlUserComponentName ((GNL_USER_COMPONENT) Compo),
            GnlUserComponentInstName ((GNL_USER_COMPONENT) Compo));
        continue;
    }

    if (Cell)
    {
        if (!(Pin = LibGetPinFromNameAndCell (Cell, (char*) Formal)))
            continue;
        if ((LibPinDirection(Pin) == OUTPUT_E) ||
            (LibPinDirection(Pin) == INOUT_E))
        {
            if (BListAddElt (*AssocSources, (int) AssocI))
                return (GNL_MEMORY_FULL);
        }
        continue;
    }

    if ( (GnlVarDir (Formal) == GNL_VAR_OUTPUT) ||
        (GnlVarDir (Formal) == GNL_VAR_INOUT) )
        if (BListAddElt (*AssocSources, (int) AssocI))
            return (GNL_MEMORY_FULL);
    }

    Function = GnlVarFunction (Var);
    if (!Function)
        return (GNL_OK);

    FunctionNode = GnlFunctionOnSet (Function);
    if (GnlNodeOp (FunctionNode) != GNL_VARIABLE)
        return (GNL_OK);

    TargetVar = (GNL_VAR) GnlNodeSons (FunctionNode);

    if (BListAddElt (*RightVarAssigneds, (int) TargetVar))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----------------------------------------------------*/
/* GnlGetDestinationsOfVar */ */
/*-----*/
GNL_STATUS GnlGetDestinationsOfVar (Var, AssocDest, LeftVarAssigneds)
    GNL_VAR Var;
    BLIST *AssocDest;
    BLIST *LeftVarAssigneds;
{
    int i;
```

gnlestim.c

```
GNL_ASSOC      AssocI;
GNL_COMPONENT   Compo;
GNL_VAR        Formal, Vari;
LIBC_PIN       Pin;
BLIST          ListAssoc;
LIBC_CELL      Cell;
GNL_STATUS     GnlStatus;

if (!Var)
    return (GNL_OK);

ListAssoc = GnlVarTraversalInfoListAssoc (Var);
if (BListCreateWithSize (1, AssocDest))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (1, LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListAssoc); i++)
{
    AssocI = (GNL_ASSOC) BListElt (ListAssoc, i);
    Compo = GnlAssocTraversalInfoComponent (AssocI);

    Vari = GnlAssocActualPort (AssocI);
    if (GnlVarIsVar (Vari))
        if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
            continue;
    Formal = GnlAssocFormalPort (AssocI);

    if (GnlStatus = GnlGetCellFromGnlCompo (Compo, &Cell))
        return (GnlStatus);

    if (GnlComponentType (Compo) == GNL_USER_COMPO &&
        !GnlUserComponentGnlDef ((GNL_USER_COMPONENT) Compo) && !Cell)
    {
        fprintf (stderr,
        " WARNING: black box <%s %s> may modify final estimation\n",
        GnlUserComponentName ((GNL_USER_COMPONENT) Compo),
        GnlUserComponentInstName ((GNL_USER_COMPONENT) Compo));
        continue;
    }

    if (Cell)
    {
        if (!(Pin = LibGetPinFromNameAndCell (Cell, (char*) Formal)))
            continue;
        if (LibPinDirection(Pin) == INPUT_E)
        {
            if (BListAddElt (*AssocDest, (int) AssocI))
                return (GNL_MEMORY_FULL);
        }
        continue;
    }

    if (GnlVarDir (Formal) == GNL_VAR_INPUT ||
        GnlVarDir (Formal) == GNL_VAR_INOUT)
```

```

        if (BListAddElt (*AssocDest, (int) AssocI))
            return (GNL_MEMORY_FULL);
    }

ListAssoc = GnlVarTraversalInfoListLeftAssign (Var);
for (i=0; i<BListSize (ListAssoc); i++)
{
    if (BListAddElt (*LeftVarAssigneds, BListElt (ListAssoc, i)))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}
/*#endif*/

/*-----
/* GnlTraversalLibCellInputs
/*-----
/* This procedure propagates thru the inputs and inouts pins of the */
/* current libc cell. We propagate thru all of them except the original */
/* pin 'Pins' (in the case of 'Pins' is an inout). */
/*-----*/
GNL_STATUS GnlTopDownTraversalVar ();
GNL_STATUS GnlTraversalLibCellInputs (UserCompo, Pins, Gnl)
    GNL_USER_COMPONENT UserCompo;
    LIBC_PIN Pins;
    GNL Gnl;
{
    int i;
    GNL_ASSOC AssocI;
    LIBC_CELL Cell;
    LIBC_PIN ListPins;
    BLIST Interface;
    char *Formal;
    GNL_VAR Actual;
    LIBC_NAME_LIST ListPinName;
    char *PinName;

    /* The libc cell. */
    Cell = GnlUserComponentCellDef (UserCompo); */

    Interface = GnlUserComponentInterface (UserCompo);

    ListPins = LibCellPins (Cell);
    for (; ListPins != NULL; ListPins = LibPinNext(ListPins))
    {
        if ((LibPinDirection (ListPins) == INPUT_E) ||
            ((LibPinDirection (ListPins) == INOUT_E) &&
            (ListPins != Pins)))
        {
            for (i=0; i<BListSize (Interface); i++)
            {
                AssocI = (GNL_ASSOC)BListElt (Interface, i);
                Formal = GnlAssocFormalPort (AssocI);
                ListPinName = LibPinName (ListPins);
            }
        }
    }
}

```

```

/* if the pin has a complex name (bus or bundle or ...) */
/* we continue. */
if (!GnlSinglePinName (ListPinName))
{
    fprintf (stderr,
              " WARNING: cell with complex names in pin\n");
    continue;
}

PinName = LibNameListName (ListPinName);
if (!strcmp (PinName, Formal))
{
    Actual = GnlAssocActualPort (AssocI);

    /* all the actual ports of an instance of a library*/
    /* cell is a simple 1-bit signal. */
    if (GnlTopDownTraversalVar (Actual, Gnl))
        return (GNL_MEMORY_FULL);
}
}

return (GNL_OK);
}

/*
 * GnlTraversalLibCell
 */
/*
 * This procedure treats the case where an association drives a libc
 * cell and is connected to it thru an output or inout pin. If it is
 * connected as an output or inout pin then we propagate recursively to
 * all the inputs + inouts of the cell. This propagation thru the inputs
 * is done in 'GnlTraversalLibCellInputs'.
 */
GNL_STATUS GnlTraversalLibCell (UserCompo, Assoc, Gnl)
    GNL_USER_COMPONENT    UserCompo;
    GNL_ASSOC             Assoc;
    GNL                  Gnl;
{
    LIBC_CELL             Cell;
    char                 *Formal;
    LIBC_PIN              Pins;
    LIBC_NAME_LIST         ListPinName;
    char                 *PinName;

    Formal = GnlAssocFormalPort (Assoc);
    Cell = GnlUserComponentCellDef (UserCompo);

    Pins = LibCellPins (Cell);

    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins);
    }
}

```

```

/* if the pin has a complex name (bus or bundle or ...) we */ 
/* continue. */ 
if (!GnlSinglePinName (ListPinName)) 
    continue;

PinName = LibNameListName (ListPinName);

if (!strcmp (PinName, Formal))
{
    /* If the net connected to the gate is accociated to an */ 
    /* input or inout pin. */ 
    if ((LibPinDirection (Pins) == OUTPUT_E) || 
        (LibPinDirection (Pins) == INOUT_E))
    {
        if (GnlTraversallibCellInputs (UserCompo, Pins, Gnl))
            return (GNL_MEMORY_FULL);
    }
    break;
}
}

return (GNL_OK);
}

/*
 * GnlTopDownTraversalVar
 */
GNL_STATUS GnlTopDownTraversalVar (Var, Gnl)
    GNL_VAR          Var;
    GNL             Gnl;
{
    int              i;
    int              j;
    BLIST           ListAssoc;
    GNL_USER_COMPONENT UserCompo;
    GNL_VAR          Formal;
    GNL_VAR          Actual;
    GNL_ASSOC         AssocI;
    GNL_VAR          SplitActualJ;
    BLIST           ListSplitActuals;
    GNL_FUNCTION     Function;
    GNL_NODE          FunctionNode;
    GNL              GnlDefCompo;
    GNL_STATUS        GnlStatus;
    GNL_VAR          TargetVar;
    GNL_USER_COMPONENT InstOfGnl;
    int              LeftFormIndex;
    int              RightFormIndex;
    int              Index;
    char             IndexFormalName;
    GNL_VAR          IndexFormalVar;

    /* List of the associations where the Var is present. */ 
    /* Warning: the var can be present as a simple var or belonging to */ 
    /* a bus: ex 'a' the GNL_VAR and the following verilog where 'toto' */ 

```

```

/* is a component, and 'ul' the instance name: */  

/*      toto ul (.x(a)); */  

/* --> the GNL_ASSOC represents the couple '(x, a)' where 'x' is the */  

/* formal and 'a' the actual parameter. */  

/* case where 'a' is used in a bus ('a' is then a 1-bit signal) */  

/*      toto ul (.x({a,b,c})); */  

/* 'x' is a [2:0] variable, {a,b,c} represents the bus formed of the */  

/* concatenation of 'a', 'b' and 'c' variables. */  

ListAssoc = GnlVarTraversalInfoListAssoc (Var);

for (i=0; i<BListSize (ListAssoc); i++)
{
    AssocI = (GNL_ASSOC)BListElt (ListAssoc, i);
    UserCompo = (GNL_USER_COMPONENT)
        GnlAssocTraversalInfoComponent (AssocI);
    Actual = GnlAssocActualPort (AssocI);

    /* actually the Formal parameter of the association is of type */
    /* char * so it is a name. We cannot go thru because the user */
    /* component should be a black box or a library cell. */
    Formal = GnlAssocFormalPort (AssocI);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
    {
        /* It corresponds to a LIBC library cell. */
        if (GnlUserComponentCellDef (UserCompo))
        {
            if (GnlTraversalLibCell (UserCompo, AssocI, Gnl))
                return (GNL_MEMORY_FULL);
        }
        else
            /* It is a black box without any definition so we stop */
            /* at its boundary. */
            {
                fprintf (stderr,
                    " WARNING: black box <%s %s> may modify final estimation\n",
                    GnlUserComponentName (UserCompo),
                    GnlUserComponentInstName (UserCompo));
            }
        continue;
    }

    /* Gnl description of the user component. */
    GnlDefCompo = GnlUserComponentGnlDef (UserCompo);

    /* If the actual var is the same as 'Var' then the formal var */
    /* is a 1-bit variable and not a bus. */
    if (GnlVarIsVar (Actual))
    {
        /* To propagate thru the component its associated Formal var*/
        /* must be an OUTPUT or an INOUT of this component. */
        if ((GnlVarDir (Formal) != GNL_VAR_OUTPUT) &&
            (GnlVarDir (Formal) != GNL_VAR_INOUT))
            continue;

        if (GnlStatus = GnlTopDownTraversalVar (Formal, Gnl))
            return (GnlStatus);
        continue;
    }
}

```

```

}

/* Otherwise 'Actual' must be a concatenation of 1-bit variables*/
/* and 'Var' must be present among them. */
ListSplitActuals = GnlNodeSons ((GNL_NODE)Actual);
LeftFormIndex = GnlVarMsb (Formal);
RightFormIndex = GnlVarLsb (Formal) ;

if (LeftFormIndex < RightFormIndex)
    Index = LeftFormIndex-1;
else
    Index = LeftFormIndex+1;

for (j=0; j<BListSize (ListSplitActuals); j++)
{
    if (LeftFormIndex < RightFormIndex)
        Index++;
    else
        Index--;
    SplitActualJ = (GNL_VAR)BListElt (ListSplitActuals, j);
    if (SplitActualJ == Var)
    {
        /* we need according to the index extract the formal */
        /* signal corresponding to this index and call recursi.*/
        /* 'GnlTopDownTraversalVar' on this signal. */
        /* with the example above we must call on: */
        /* 'GnlTopDownTraversalVar (x[2])' if 'a' is 'Var'. */

        /* 'Index' is the value of the formal index bit */
        /* that we expect. */

        /* 'IndexFormalName' is a new string in memory. */
        if (GnlVarIndexName (Formal, Index, &IndexFormalName))
            return (GNL_MEMORY_FULL);

        if ((GnlStatus = GnlGetVarFromName (GnlDefCompo,
                                            IndexFormalName, &IndexFormalVar)))
        {
            if (GnlStatus == GNL_VAR_NOT_EXISTS)
            {
                fprintf (stderr, " ERROR: cannot find var\n");
                return (GNL_VAR_NOT_EXISTS);
            }
            return (GNL_MEMORY_FULL);
        }

        if (GnlStatus = GnlTopDownTraversalVar (IndexFormalVar,
                                                Gnl))
            return (GnlStatus);
        break;
    }
}

/* we traverse now the target Var of the current 'Var'. There is */
/* only 1 target var if not there is a multisource problem. */
/* if 'Var' is 'a' and we have the following verilog example: */

```

gnlestim.c

```
/*
 * assign a = x;
 * then we need to propagate thru 'x'.
 * We stay in the same Gnl.
 * Function = GnlVarFunction (Var);
 */

if (!Function)
    return (GNL_OK);

FunctionNode = GnlFunctionOnSet (Function);
if (GnlNodeOp (FunctionNode) == GNL_CONSTANTE)
    return (GNL_OK);

if (GnlNodeOp (FunctionNode) != GNL_VARIABLE)
{
    fprintf (stderr,
" WARNING: the traversal passes thru a real boolean function on net <%s>\n",
        GnlVarName (Var));
    return (GNL_OK);
}

/* So we have a function on 'Var' of the form : 'Var = TargetVar'. */
TargetVar = (GNL_VAR)GnlNodeSons (FunctionNode);

/* we propagate the traversal on the 'TargetVar'. */
if ((GnlStatus = GnlTopDownTraversalVar (TargetVar, Gnl)))
    return (GnlStatus);

return (GNL_OK);
}

/*
 * GnlTopDownTraversalGnl
 */
GNL_STATUS GnlTopDownTraversalGnl (Gnl)
    GNL          Gnl;
{
    int           i;
    int           j;
    GNL_STATUS    GnlStatus;
    BLIST         BucketI;
    GNL_VAR       VarJ;
    BLIST         HashTableNames;

/* we scan teh Hash tables of the current 'Gnl' to access all its
 * variables (nets).
 */
HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);

        /* If the net is a primary output or inout then we start */
    }
}
```

gnlestim.c

```
/* to traverse in a top down manner. */  
if ((GnlVarDir (VarJ) == GNL_VAR_OUTPUT) ||  
    (GnlVarDir (VarJ) == GNL_VAR_INOUT))  
{  
    if ((GnlStatus = GnlTopDownTraversalVar (VarJ, Gnl)))  
        return (GnlStatus);  
}  
}  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlTopDownTraversal */  
/*-----*/  
GNL_STATUS GnlTopDownTraversalNw (Nw)  
    GNL_NETWORK Nw;  
{  
    GNL          TopGnl;  
    GNL_STATUS   GnlStatus;  
  
    TopGnl = GnlNetworkTopGnl (Nw);  
  
    if ((GnlStatus = GnlTopDownTraversalGnl (TopGnl)))  
        return (GnlStatus);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlSetListPathComponentInGnl */  
/*-----*/  
GNL_STATUS GnlSetListPathComponentInGnl (Gnl, PreviousPathCompo)  
    GNL          Gnl;  
    GNL_PATH_COMPONENT PreviousPathCompo;  
{  
    int          j;  
    BLIST       Components;  
    GNL_COMPONENT ComponentJ;  
    GNL_USER_COMPONENT UserCompoJ;  
    GNL          Gnldef;  
    GNL_PATH_COMPONENT newPathCompo;  
    GNL_USER_COMPONENT UserCompo;  
    char         *InstanceName;  
    BLIST       SubList;  
    unsigned int Key;  
  
    if (PreviousPathCompo)  
    {  
        UserCompo = GnlPathComponentComponent (PreviousPathCompo);  
        InstanceName = GnlUserComponentInstName (UserCompo);  
        Key = KeyOfName (InstanceName,  
                         BListSize (GnlListPathComponent (Gnl)));  
    }
```

gnlestim.c

```
SubList = (BLIST)BListElt (GnlListPathComponent (Gnl), Key);
if (!SubList)
{
    if (BListCreateWithSize (1, &SubList))
        return (GNL_MEMORY_FULL);
    BListElt (GnlListPathComponent (Gnl), Key) = (int)SubList;
}

if (BListAddElt (SubList, (int)PreviousPathCompo))
    return (GNL_MEMORY_FULL);

}

Components = GnlComponents (Gnl);
if (!Components)
    return (GNL_OK);

for (j=0; j<BListSize (Components); j++)
{
    ComponentJ = (GNL_COMPONENT)BListElt (Components, j);
    if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)
        continue;

    UserCompoJ = (GNL_USER_COMPONENT)ComponentJ;

    /* Apparently a black box or a LIBC cell */
    if (!GnlUserComponentGnlDef (UserCompoJ))
        continue;

    /* we create a new Path component storing the current component */
    if (GnlCreatePathComponent (& newPathCompo))
        return (GNL_MEMORY_FULL);
    SetGnlPathComponentComponent (NewPathCompo, UserCompoJ);
    /* We point on the previous Path component */
    SetGnlPathComponentPrevious (NewPathCompo, PreviousPathCompo);

    Gnldef = GnlUserComponentGnlDef (UserCompoJ);

    if (GnlSetListPathComponentInGnl (Gnldef, newPathCompo))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlPrintPathComponent
/*-----*/
void GnlPrintPathComponent (PathCompo)
    GNL_PATH_COMPONENT    PathCompo;
{
    GNL_PATH_COMPONENT    Previous;
    GNL_USER_COMPONENT    CompoI;

    if (!PathCompo)
        return;
```

```

Previous = GnlPathComponentPrevious (PathCompo);
GnlPrintPathComponent (Previous);

CompoI = GnlPathComponentComponent (PathCompo);
fprintf (stderr, "%s.", GnlUserComponentInstName (CompoI));
}

/*
 * GnlPrintListPathComponentInGnl
 */
GNL_STATUS GnlPrintListPathComponentInGnl (Nw, Gnl)
    GNL_NETWORK          Nw;
    GNL                  Gnl;
{
    int                 i;
    int                 j;
    BLIST               Components;
    GNL_COMPONENT        ComponentJ;
    GNL_USER_COMPONENT   UserCompoJ;
    GNL                 Gnldef;
    GNL_PATH_COMPONENT  PathCompoI;

/* Already traversed and considered this 'Gnl'. */
if (GnlTag (Gnl) == GnlNetworkTag (Nw))
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

fprintf (stderr, " GNL = <%s>\n", GnlName (Gnl));
for (i=0; i<BListSize (GnlListPathComponent (Gnl)); i++)
{
    fprintf (stderr, " ");
    PathCompoI = (GNL_PATH_COMPONENT)BListElt (
                    GnlListPathComponent (Gnl), i);
    GnlPrintPathComponent (PathCompoI);
    fprintf (stderr, "\n");
}
fprintf (stderr, "\n");

Components = GnlComponents (Gnl);
if (!Components)
    return (GNL_OK);

for (j=0; j<BListSize (Components); j++)
{
    ComponentJ = (GNL_COMPONENT)BListElt (Components, j);
    if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)
        continue;

    UserCompoJ = (GNL_USER_COMPONENT)ComponentJ;

/* Apparently a black box or a LIBC cell */
if (!GnlUserComponentGnlDef (UserCompoJ))
    continue;
*/
}

```

```

Gnldef = GnlUserComponentGnlDef (UserCompoJ);

if (GnlPrintListPathComponentInGnl (Nw, Gnldef))
    return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
/* GnlSetListPathComponent
*/
/*
/* This procedure scans all the Gnl from the top gnl of the network */
/* and sets the list path components in each of them. */
*/
GNL_STATUS GnlSetListPathComponent (Nw)
    GNL_NETWORK Nw;
{
    GNL      TopGnl;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (GnlSetListPathComponentInGnl (TopGnl, NULL))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
/* GnlPrintListPathComponent
*/
/*
void GnlPrintListPathComponent (Nw)
    GNL_NETWORK Nw;
{
    GNL  TopGnl;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    TopGnl = GnlNetworkTopGnl (Nw);

    GnlPrintListPathComponentInGnl (TopGnl);
}

/*
/* GnlResizeHashListPathComponentInGnl
*/
/*
#define TRACE_RESIZING
GNL_STATUS GnlResizeHashListPathComponentInGnl (Gnl)
    GNL Gnl;
{
    int          i;
    int          NbInstances;
    BLIST       SubList;
}

```

gnlestim.c

```
int                  NewSize;
BLIST      NewHashList;
GNL_PATH_COMPONENT   PathJ;
int                  j;
unsigned int         Key;
BLIST      NewSubList;
GNL_USER_COMPONENT  UserCompoJ;
char                 *InstanceNameJ;
int                  MaxSubList;

NbInstances = 0;
for (i=0; i<BListSize (GnlListPathComponent (Gnl)); i++)
{
    SubList = (BLIST)BListElt (GnlListPathComponent (Gnl), i);
    if (SubList)
        NbInstances += BListSize (SubList);
}

#ifndef TRACE_RESIZING
fprintf (stderr,
        " <%s> Nb.Inst=%d\n", GnlName (Gnl), NbInstances);
#endif

if (NbInstances <= HASH_LIST_PATH_COMPO_SIZE)
    return (GNL_OK);

#ifndef TRACE_RESIZING
fprintf (stderr,
        " Resizing Hash table path compo of <%s>[Nb.Inst=%d]\n",
        GnlName (Gnl), NbInstances);
#endif

/* We want too have about not more than 1 elements in each sublist */
/* of the hash list. */
NewSize = NbInstances+1;
if (BListCreateWithSize (NewSize, &NewHashList))
    return (GNL_MEMORY_FULL);

BSize (NewHashList) = NewSize;

for (i=0; i<BListSize (GnlListPathComponent (Gnl)); i++)
{
    SubList = (BLIST)BListElt (GnlListPathComponent (Gnl), i);
    if (!SubList)
        continue;

    for (j=0; j<BListSize (SubList); j++)
    {
        PathJ = (GNL_PATH_COMPONENT)BListElt (SubList, j);
        UserCompoJ = GnlPathComponentComponent (PathJ);
        InstanceNameJ = GnlUserComponentInstName (UserCompoJ);
        Key = KeyOfName (InstanceNameJ, BListSize (NewHashList));
        NewSubList = (BLIST)BListElt (NewHashList, Key);
        if (!NewSubList)
        {
```

```

        if (BListCreateWithSize (1, &NewSubList))
            return (GNL_MEMORY_FULL);
        BListElt (NewHashList, Key) = (int)NewSubList;
    }

    if (BListAddElt (NewSubList, (int)PathJ))
        return (GNL_MEMORY_FULL);
}

BListQuickDelete (&SubList);
}

BListQuickDelete (&GnlListPathComponent (Gnl));
SetGnlListPathComponent (Gnl, NewHashList);

#endif TRACE_RESIZING
MaxSubList = 0;
for (i=0; i<BListSize (NewHashList); i++)
{
    SubList = (BLIST)BListElt (NewHashList, i);
    if (BListSize (SubList) > MaxSubList)
        MaxSubList = BListSize (SubList);
}
fprintf (stderr, " Max SubList = %d\n", MaxSubList);
#endif

return (GNL_OK);
}

/*
 * GnlResizeHashListPathComponentInGnlRec
 */
/*
 * This procedure resize the hash list of path component depending on
 * the number of path component in the hash list. If the number of path
 * component is greater than HASH_LIST_PATH_COMPO_SIZE then we resize
 * it in order to expect about 1 element in each sublist.
 */
GNL_STATUS GnlResizeHashListPathComponentInGnlRec (Nw, Gnl)
    GNL_NETWORK      Nw;
    GNL             Gnl;
{
    int                  NbInstances;
    int                  i;
    int                  j;
    GNL                 Gnldef;
    BLIST               Components;
    GNL_COMPONENT       ComponentJ;
    GNL_USER_COMPONENT UserCompoJ;

    /* If already done then we return. */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));
}

```

gnlestim.c

```
if (GnlResizeHashListPathComponentInGnl (Gnl))
    return (GNL_MEMORY_FULL);

/* We do this recursively ...
Components = GnlComponents (Gnl);
if (!Components)
    return (GNL_OK);

for (j=0; j<BListSize (Components); j++)
{
    ComponentJ = (GNL_COMPONENT)BListElt (Components, j);
    if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)
        continue;

    UserCompoJ = (GNL_USER_COMPONENT)ComponentJ;

    /* Apparently a black box or a LIBC cell */
    if (!GnlUserComponentGnlDef (UserCompoJ))
        continue;

    Gnldef = GnlUserComponentGnlDef (UserCompoJ);

    if (GnlResizeHashListPathComponentInGnlRec (Nw, Gnldef))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
/* GnlResizeHashListPathComponent
*/
/*
/* we resize the hash list of path component depending on the number
/* of instances leading to a current Gnl.
*/
/*
GNL_STATUS GnlResizeHashListPathComponent (Nw)
    GNL_NETWORK Nw;
{
    GNL TopGnl;

    /* Resetting the Tag for recursive traversal.
     * SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    TopGnl = GnlNetworkTopGnl (Nw);

    if (GnlResizeHashListPathComponentInGnlRec (Nw, TopGnl))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
/* GnlTimingEstimate
*/
/*
/* This procedure computes the timing estimations related to the global */
```

gnlestim.c

```
/* netlist 'GlobalNetwork' using library cells of 'GnlLibc'. */  
/*-----*/  
GNL_STATUS GnlTimingEstimate (Nw, GnlLibc)  
    GNL_NETWORK    Nw;  
    LIBC_LIB GnlLibc;  
{  
    GNL                      TopGnl;  
    GNL_STATUS                GnlStatus;  
    int                       RespectLibConstraints;  
    int                       PrintClkDomain;  
    int                       PrintCritRegion;  
  
    /* Set up the list of path components to avoid the brutal flattening */  
    /* during estimation. */  
    fprintf (stderr, " Performing Timing Analysis...\n");  
  
    if (!GnlEnvRespectLibConstraints() && !GnlEnvPostOpt () &&  
        (GnlEnvMode () != GNL_MODE_POST_OPTIMIZATION) )  
    {  
        if (GnlSetListPathComponent (Nw))  
            return (GNL_MEMORY_FULL);  
  
        /* we resize the hash list of path component depending on the number */  
        /* of instances leading to a current Gnl. */  
        if (GnlResizeHashListPathComponent (Nw))  
            return (GNL_MEMORY_FULL);  
  
        #ifdef TRACE_PATH  
            GnlPrintListPathComponent (Nw);  
        #endif  
    }  
    /* Resetting the Tag for recursive traversal. */  
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);  
  
    TopGnl = GnlNetworkTopGnl (Nw);  
  
    PrintClkDomain = GnlEnvPrintClkDomain();  
    PrintCritRegion = GnlEnvPrintCritRegion();  
  
    if (TimeEstimate (Nw, GnlLibc, PrintClkDomain, PrintCritRegion))  
        return (GNL_MEMORY_FULL);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlTimingEstimateAfterSynthesis */  
/*-----*/  
/* This procedure computes the timing estimations related to the global */  
/* netlist 'GlobalNetwork' using library cells of 'GnlLibc'. */  
/*-----*/  
GNL_STATUS GnlTimingEstimateAfterSynthesis (Nw, GnlLibc)  
    GNL_NETWORK    Nw;  
    LIBC_LIB GnlLibc;  
{
```

gnlestim.c

```
GNL          TopGnl;
GNL_STATUS   GnlStatus;

TopGnl = GnlNetworkTopGnl (Nw);
if (GnLinkLib (Nw, TopGnl, GnlLibc))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlTimingEstimate (Nw, GnlLibc)))
    return (GnlStatus);

return (GNL_OK);
}

/*-----*/
/* GnlEstimate */
/*-----*/
GNL_STATUS GnlEstimate ()
{
    int             i;
    GNL            Gnl;
    GNL            GnlI;
    BLIST          ListGnls;
    LIBC_LIB       GnlLib;
    GNL_LIB        HGnlLib;
    int            Done;
    FILE           *OutFile;
    GNL_STATUS     GnlStatus;
    GNL            TopGnl;
    GNL_NETWORK   GlobalNetwork;

    if (GnlEnvOutput ())
    {
        if ((OutFile = fopen (GnlEnvOutput (), "w")) == NULL)
        {
            fprintf (stderr, " ERROR: Cannot Open Output File '%s'\n",
                     GnlEnvOutput ());
            return (GNL_CANNOT_OPEN_OUTFILE);
        }
        fclose (OutFile);
    }

    if ((OutFile = fopen (GnlEnvLib (), "r")) == NULL)
    {
        fprintf (stderr, " ERROR: Cannot Open Library File '%s'\n",
                 GnlEnvLib ());
        return (GNL_CANNOT_OPEN_LIBRARY);
    }
    fclose (OutFile);
    if (GnlEnvInputFormat () == GNL_INPUT_VLG)
    {
        fprintf (stderr, "\n Reading Verilog file [%s] ... \n",
                 GnlEnvInput ());
        if (GnlRead (GnlEnvInput (), &ListGnls))
        {
```

```

        fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
                  GnlEnvInput());
        return (GNL_CANNOT_OPEN_INPUTFILE);
    }
    fprintf (stderr, " Verilog Netlist Analyzed\n");
}
else
{
    fprintf (stderr,
            " ERROR: Estimation must be done on Verilog netlist input\n");
    return (GNL_CANNOT_OPEN_INPUTFILE);
}

/* Sort and prints the list 'ListGnls' by putting first the top          */
/* level modules                                                       */
SortTopLevelModules (ListGnls);
GnlPrintTopLevels (ListGnls);

if (!GnlEnvTop())
{
    GnlI = (GNL)BListElt (ListGnls, 0);
    TopGnl = GnlI;
    fprintf (stderr,
            " WARNING: Top Level Module is [%s] by default\n",
            GnlName (TopGnl));
}
else
{
    /* Taking the top level module from the user                      */
    TopGnl = NULL;
    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnlI = (GNL)BListElt (ListGnls, i);
        if (!strcmp (GnlName (GnlI), GnlEnvTop()))
        {
            TopGnl = GnlI;
            break;
        }
    }
}

if (TopGnl == NULL)
{
    fprintf (stderr, " ERROR: cannot find top-level module [%s]\n",
            GnlEnvTop());
    return (GNL_CANNOT_FIND_TOP_LEVEL);
}

/* we create the Global network.                                         */
if (GnlCreateNetwork (TopGnl, &GlobalNetwork))
    return (GNL_MEMORY_FULL);

/* Reading and building the technology library                         */
fprintf (stderr, "\n Reading Library [%s] ...\n",
        GnlEnvLib());
if (LibRead (GnlEnvLib(), &GnlLib))

```

```

{
    fprintf (stderr, " ERROR: Library analysis aborted\n");
    return (GNL_ERROR_LIBRARY_READING);
}
fprintf (stderr, " Library analyzed\n");

/* Checking correctness of the network 'GlobalNetwork'. */
if ((GnlStatus = GnlCheckNetwork (GlobalNetwork)))
    return (GnlStatus);

fprintf (stderr,
        "\n Checking Hierachical interfaces from top module [%s]\n",
        GnlName (TopGnl));

/* We verify the correct connection between user components and */
/* their definitions and update some fields (RefCount). */
/* if (GnlUpdateNCheckHierarchyInterface (GlobalNetwork, TopGnl, */
/*                                         ListGnls))
    return (GNL_MEMORY_FULL);

/* We link each component of all the Gnls in the network with the */
/* libc cells of same name. */
fprintf (stderr, " Linking Components With Libc Cells\n");
if (GnlLinkLib (GlobalNetwork, TopGnl, GnlLib))
    return (GNL_MEMORY_FULL);

/* We modify the original GNL_USER_COMPONENT components according to */
/* to their linking libc cell. Only Predefined components like Dffs, */
/* latches, tristates are replaced. */
/* if (GnlReplacePredefComponentsWithLinkLibCells (GlobalNetwork, TopGnl, */
/*                                               GnlLib))
    return (GNL_MEMORY_FULL);

if (GnlReportDataForEstimation (GlobalNetwork, GnlLib))
    return (GnlStatus);

return (GNL_OK);
}
/*-----*/

```

gnlestim.h

```
/*-----*/
/*          gnlestim.h           */
/*-----*/
/* File:      gnlestim.h           */
/* Version:   1.1                  */
/* Modifications: -                */
/* Documentation: -                */
/*-----*/
#ifndef GNLESTIM_H
#define GNLESTIM_H

/*-----*/
/* Info stored in GNL_ASSOC thru Hook field           */
/*-----*/
typedef struct GNL_ASSOC_TRAVERSAL_INFO_STRUCT {
    GNL_COMPONENT Component; /* Component to which the assoc        */
                           /* belongs to.                         */
    void             *Hook;
} GNL_ASSOC_TRAVERSAL_INFO_REC, *GNL_ASSOC_TRAVERSAL_INFO;

#define GnlAssocTraversalInfoComponent(n) \
    (((GNL_ASSOC_TRAVERSAL_INFO)((n)->Hook))->Component)
#define GnlAssocTraversalInfoHook(n) \
    (((GNL_ASSOC_TRAVERSAL_INFO)((n)->Hook))->Hook)

#define SetGnlAssocTraversalInfoComponent(n,b) \
    (((GNL_ASSOC_TRAVERSAL_INFO)((n)->Hook))->Component = b)
#define SetGnlAssocTraversalInfoHook(n,b) \
    (((GNL_ASSOC_TRAVERSAL_INFO)((n)->Hook))->Hook = b)

/*-----*/
/* Info stored in GNL_VAR thru Hook field           */
/*-----*/
typedef struct GNL_VAR_TRAVERSAL_INFO_STRUCT {
    BLIST ListAssoc; /* List of association where the var is   */
                      /* present in the actual parameter.       */
    BLIST ListLeftAssign; /* List of GNL_VAR which are on the left */
                          /* assignment of the current var. */
                          /* assign x = y; where 'y' current var. */
    void             *Hook;
} GNL_VAR_TRAVERSAL_INFO_REC, *GNL_VAR_TRAVERSAL_INFO;

#define GnlVarTraversalInfoListAssoc(n) \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->ListAssoc)
#define GnlVarTraversalInfoListLeftAssign(n) \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->ListLeftAssign)
#define GnlVarTraversalInfoHook(n) \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->Hook)

#define SetGnlVarTraversalInfoListAssoc(n,b) \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->ListAssoc = b)
#define SetGnlVarTraversalInfoListLeftAssign(n,b) \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->ListLeftAssign = b)
```

gnlestim.h

```
#define SetGnlVarTraversalInfoHook(n, b)      \
    (((GNL_VAR_TRAVERSAL_INFO)((n)->Hook))->Hook = b)

/* ----- EOF ----- */

#endif
```

gnlfsm.h

```
/*-----*/  
/*-----*/  
/*      File:          gnlfsm.h           */  
/*      Version:        1.1             */  
/*      Modifications: -            */  
/*      Documentation: -           */  
/*-----*/  
/*-----*/  
/*      Class: none           */  
/*      Inheritance:          */  
/*-----*/  
/*-----*/  
  
/*-----*/  
/* DEFINES for Dead code detection           */  
/*-----*/  
  
#define DCD_Unknown_Region          0  
#define DCD_Architecture_Body      1  
#define DCD_Subprogram_Body         2  
#define DCD_Guarded_Assignment    3  
#define DCD_If_Region               4  
#define DCD_Elsif_Region            5  
#define DCD_Else_Region              6  
#define DCD_CaseWhen_Region         7  
#define DCD_CaseWhenOthers_Region   8  
#define DCD_Loop_Region             9  
#define DCD_Next_Branchmnt         10  
#define DCD_Exit_Branchmnt         11  
#define DCD_After_Next             12  
#define DCD_After_Exit              13  
#define DCD_After_Return            14  
#define DCD_After_If                15  
#define DCD_After_Case              16  
#define DCD_After_Wait              17  
#define DCD_Empty_Region            18  
#define DCD_Waveform                19  
#define DCD_Else_Waveform           21  
#define DCD_Others_Waveform         21
```

gnlfsm.l

```
%{
/*-----*/  
/*-----*/  
/*      File:          gnlfsm.l */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
  
#include <string.h>  
#include <ctype.h>  
#include <string.h>  
  
#include "fsmtokens.h"  
#include "gnlfsm.h"  
  
/*-----*/  
int      yylex           (void);  
int      yylook          (void);  
int      yyreject        (void);  
int      yywinput         (void);  
int      yyinput          (void);  
int      ymbinput        (void);  
char *   mbsinvalid     (const char *s); /* defined in mbstr.h */  
  
/* Text Buffer in LEX */  
# undef YYLMAX  
# define YYLMAX (MAXNAMELENGTH + 512)  
  
# define YY_NO_UNPUT      /* to remove definition of yyunput */  
  
extern int  LineNumberReadFsm;           /* Input Stream line counter */  
extern FILE *yyin;  
extern int  yydebug;  
extern int  FsmVersion ;               /* Fsm version */  
  
%}  
%START ATTR V3 V4 V5  
%p 7650  
%n 1300  
%a 9000  
%o 8500  
%e 1550  
%%  
^.*\n{  
# ifdef MYLEXDEBUG  
#   fprintf (yyout, "\n%4d %s",  
#             LineNumberReadFsm, yytext);  
# endif  
#
```

gnlfsm.l

```
        REJECT;
    }

^".fsmversion"[ ]+[0-9]+ { /* .fsmversion 5 */
    char *pc = strrchr (yytext, ' ');
    #ifdef MYLEXDEBUG
        fprintf (yyout, "FSMVERSION %s", yytext);
    #endif
    sscanf (pc, "%d", &FsmVersion);
    switch (FsmVersion) {
        case 3: BEGIN V3; break;
        case 4: BEGIN V4; break;
        case 5: BEGIN V5; break;
    default: BEGIN V3;
    }
    #ifdef MYLEXDEBUG
        fprintf (yyout, " FsmVersion=%d\n",
        FsmVersion);
    #endif
    return FSMVERSION;
}

^".date "
#
#
{
    #ifdef MYLEXDEBUG
        fprintf (yyout, "DATE\n");
    #endif
    if (FsmVersion == 3) BEGIN V3;
    return DATE;
}

^".signature "
^".paramvalue "
^".returnrange "
^".pragma "
^".fsm "
^".library "
^".instances "
^".clocks"
^".inputs"
^".outbuses"
^".inoutbuses"
^".statevars"
^".breakpoints"
^".bbinputs"
^".bbinouts"
^".bboutputs"
^".bbgenerics"
^".dontcares"
^".globals"
^".axioms"
^".asserts"
^".valid "
^".values"
^".functions"
^".end"
^".eof"
^".history"
^".OriginLang"
{
    return SIGNATURE;
    return PARAMVALUE;
    return RETURNRANGE;
    return PRAGMA;
    return FSM;
    return LIBRARY;
    return INSTANCES;
    return CLOCKS;
    return INPUTS;
    return OUTBUSES;
    return INOUTBUSES;
    return STATEVARS;
    return BREAKPOINTS;
    return BBINPUTS;
    return BBINOUTS;
    return BBOUTPUTS;
    return BBGENERICS;
    return DONTCARES;
    return GLOBALS;
    return AXIOMS;
    return ASSERTS;
    return VALID;
    return VALUES;
    return FUNCTIONS;
    return END;
    return EOF;
    return HISTORY;
    return ORIGINLANG;
}
```

gnl fsm.l

```
^".filenames"          { return FILENAMES;           }
^".deadcodedetects"   { return DEADCODEDETECT;      }
".loc"                 { return LOC;                  }
".altname"             { return ALTNAMESPACE;        }
".not"                 { return NOT;                  }
".type"                { return TYPE;                 }
^".enumerations"       { return ENUMERATIONS;        }
".enum_encoding"       { return ENUM_ENCODING;       }
[()]                  { return OPENPAR;              }
[]                     { return CLOSEPAR;             }
[:]                     { BEGIN ATTR; return COLON;  }
[,]                     { return COMMA;                }
[-]                     { return DASH;                 }
[/]                     { return SLASH;                }

"@Unknown_Region"      { yyval.integ = DCD_Unknown_Region;
                        return DCD_REGION;
}
"@Architecture_Body"   { yyval.integ = DCD_Architecture_Body;
                        return DCD_REGION;
}
"@Subprogram_Body"     { yyval.integ = DCD_Subprogram_Body;
                        return DCD_REGION;
}
"@Guarded_Assignment" { yyval.integ = DCD_Guarded_Assignment;
                        return DCD_REGION;
}
"@If_Region"            { yyval.integ = DCD_If_Region;
                        return DCD_REGION;
}
"@Elsif_Region"         { yyval.integ = DCD_Elsif_Region;
                        return DCD_REGION;
}
"@Else_Region"          { yyval.integ = DCD_Else_Region;
                        return DCD_REGION;
}
"@CaseWhen_Region"      { yyval.integ = DCD_CaseWhen_Region;
                        return DCD_REGION;
}
"@CaseWhenOthers_Region" { yyval.integ = DCD_CaseWhenOthers_Region;
                        return DCD_REGION;
}
"@Loop_Region"           { yyval.integ = DCD_Loop_Region;
                        return DCD_REGION;
}
"@Next_Jump"             { yyval.integ = DCD_Next_Branchmnt;
                        return DCD_REGION;
}
"@Exit_Jump"              { yyval.integ = DCD_Exit_Branchmnt;
                        return DCD_REGION;
}
"@After_Next"             { yyval.integ = DCD_After_Next;
                        return DCD_REGION;
}
"@After_Exit"              { yyval.integ = DCD_After_Exit;
                        return DCD_REGION;
}
```

gnl fsm.l

```
"@After_Return"
    { yyval.integ = DCD_After_Return;
return DCD_REGION;
}
"@After_If"
    { yyval.integ = DCD_After_If;
return DCD_REGION;
}
"@After_Case"
    { yyval.integ = DCD_After_Case;
return DCD_REGION;
}
"@After_Wait"
    { yyval.integ = DCD_After_Wait;
return DCD_REGION;
}
"@Empty_Region"
    { yyval.integ = DCD_Empty_Region;
return DCD_REGION;
}
"@Waveform"
    { yyval.integ = DCD_Waveform;
return DCD_REGION;
}
"@Else_Waveform"
    { yyval.integ = DCD_Else_Waveform;
return DCD_REGION;
}
"@Others_Waveform"
    { yyval.integ = DCD_Others_Waveform;
return DCD_REGION;
}
<ATTR>"Enumerate"
{
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "ENUMERATE %s\n", yytext);
    #endif
    switch (FsmVersion) {
        case 3: BEGIN V3; break;
        case 4: BEGIN V4; break;
        case 5: BEGIN V5; break;
    default: BEGIN V3;
    }
    return ENUMERATE;
}
<ATTR>"Integer"
{
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "KINTEGER %s\n", yytext);
    #endif
    switch (FsmVersion) {
        case 3: BEGIN V3; break;
        case 4: BEGIN V4; break;
        case 5: BEGIN V5; break;
    default: BEGIN V3;
    }
    return KINTEGER;
}
".Options "
".Options "
[.]
{
    /* hierarchical '..' */
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "HPOINT %s\n", yytext);
    #endif
}
```

gnl fsm.l

```

#
#                                endif
#                                return HPOINT;
}

["] (["] ["]" | [^"]) *["]
{
    /* "abc""def" -> string abc"def */
    char *str, *pc;
    int strlg = strlen (yytext)-2;
    yylval.string = (char*)malloc (strlg+1);
    str = yylval.string;
    pc = yytext + 1; /* sauter le 1er '"' */
    while (*pc != '\0') {
        if (*pc == '\n') LineNumberReadFsm++;
        if (*pc == '"' && *(pc+1) == '"') pc++;
        *str = *pc;
        str++; pc++;
    }
    /* supprimer la derniere '"' */
    *(--str) = '\0';
    ifdef MYLEXDEBUG
        fprintf (yyout, "STRING
%s\n",yylval.string);
    #
    endif
    return STRING;
}

['] [\\"\\] [\\"\\] [']
['].[']
{
    /* 'a', '*', '\\"' */
    yylval.string = strdup (yytext);
    ifdef MYLEXDEBUG
        fprintf (yyout, "CHARACT %s\n", yytext);
    endif
    return CHARACT;
}

<V4>vf [:] [0-9]+[() [0-9]+[]]
<V5>vf [@] [0-9]+[() [0-9]+[]]
[\\"\\] ([\\\"] | [^\\\"])*[\\"\\] ([()-? [a-zA-Z0-9]+([,-? [a-zA-Z0-9]+)*[]])+
[a-zA-Z] [a-zA-Z0-9_]*( [()-? [a-zA-Z0-9]+([,-? [a-zA-Z0-9]+)*[]])+
/* ab(3,52)(2), \ab\nc\(3,52)(2) */
yylval.string = strdup (yytext);
ifdef MYLEXDEBUG
    fprintf (yyout, "BITIDENT %s\n", yytext);
endif
return BITIDENT;
}

['] [0-9]+
#
yytext,
{
    /* '12 '334 -> 12 334 */
    sscanf (yytext+1, "%d", &yylval.integ);
    ifdef MYLEXDEBUG
        fprintf (yyout, "TICKNUM %s, %d\n",
yylval.integ);
    endif
    return TICKNUM;
}

[']stable
|

```

gnl fsm.l

```
[']event { /* 'event or 'stable */
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "TICKEV %s\n", yytext);
    #endif
    return TICKEV;
}

([()(-?[0-9]+(" downto "|" to ")-?[0-9]+([,]-?[0-9]+(" downto "|" to ")-?[0-9]+)*[])])+
{ /* (5 downto -2,-7 to 10)(2 to 8)(13 downto 9)
*/
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "RANGE %s\n", yytext);
    #endif
    return RANGE;
}

<V4>(:)rcf_+
<V3,V5>(@)rcf_+
{ /* V3 for bpsubst, fsmr, has re-written an fsmV3 into an fsmV5 */
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "RECORDFIELD %s\n",
yytext);
    #endif
    return RECORDFIELD;
}

<V4>mod[crv][:]
<V5>mod[crv] @@
<V4>mo[dfprw][:]
<V5>mo[dfprw] @@
<V4>ano[:]
<V5>ano @@
<V4>a[0-9]+o[:]
<V5>a[0-9]+o @@
<V4>bp[0-9]+[:]
<V5>bp[0-9]+ @@
<V4>line[0-9]+[:]
<V5>line[0-9]+ @@
{ /* PREFIX */
    yyval.string = strdup (yytext);
    #ifdef MYLEXDEBUG
        fprintf (yyout, "PREFIX %s\n", yytext);
    #endif
    return PREFIX;
}

mc[@|][']event
<V4>dc_line[0-9]+[:] [0-9]+
<V5>dc_line[0-9]+[@] [0-9]+
<V4>vf[:] [0-9]+
<V5>vf[@] [0-9]+
[\\\]([\\\][\\]|[^\\])*[\\\]
<V3,V4,V5>[a-zA-Z][a-zA-Z0-9_]* { /* \ab\\c25\, abc25 */
```

gnl fsm.l

```
yyval.string = strdup (yytext);
#
#       ifdef MYLEXDEBUG
#           fprintf (yyout, "ID %s\n", yytext);
#       endif
#       return ID;
#   }

-? [0-9]+          { /* 10, -73 */
#       sscanf (yytext, "%d", &yyval.integ);
#       ifdef MYLEXDEBUG
#           fprintf (yyout, "INTEGER %s, %d\n",
#                     yytext, yyval.integ);
#       endif
#       switch (FsmVersion) {
#       case 3: BEGIN V3; break;
#       case 4: BEGIN V4; break;
#       case 5: BEGIN V5; break;
#       default: BEGIN V3;
#       }
#       return INTEGER;
#   }

"--".*
";"
"#".*
";"
\n
{
    LineNumberReadFsm++;
}

['_']
";"
"\n"
;<<EOF>>
\{fprintf(stdout, "\n"); return 0; \}
\t
/* Last LEX rule */

#if defined(FLEX_SCANNER)
# ifdef CHECK_ALLOC
    /* Avoid trap malloc of internal flex-2.5.2 code */
#   undef malloc
#   undef calloc
#   undef realloc
#   undef strdup
#   undef free
# endif
#endif
}

%%

#if defined(FLEX_SCANNER)
# ifdef CHECK_ALLOC
    /* Re-trap malloc for user code (in the case of flex-2.5.2) */
#   define malloc      MALLOC
#   define calloc     CALLOC
#   define realloc    REALLOC
#   define strdup    STRDUP
#   define free      FREE
# endif
#endif
```

gnl fsm.l

```
int yylexclearin (void)
{
    int token;
    while ((token = yylex ()) != END) {
        switch (token) {
        case STRING:
        case BITIDENT:
        case ID:
        case RANGE:
        case ENUMERATE:
        case KINTEGER:
        case HPOINT:
        case CHARACT:
        case TICKEV:
        case PREFIX:
        case RECORDFIELD:
            free (yylval.string);
            break;
        default: break;
        }
    }
    return 0;
}
```

gnlfsm.y

```
%{
/*-----*/  
/*-----*/  
/*      File:          gnlfsm.y */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
  
#include <stdio.h>  
#include <ctype.h>  
#include <malloc.h>  
#include <string.h>  
  
#include "blist.h"  
#include "gnl.h"  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern BLIST      G_ListOfGnls;  
extern GNL      G_CurrentGnl;  
  
/*-----*/  
/* GLOBAL */  
/*-----*/  
GNL_STATUS  G_GnlStatus;  
GNL_VAR      G_Var;  
char         *G_NewName;  
int          G_NbBddNodes;  
  
BLIST       G_HashListFsmVar; /* Hash list where fsm var. are      */  
                  /* hashed according to their Id      */  
BLIST       G_ListFuncIdDontCare; /* List of BitIdent corresponding to  
                  /* axioms that we will ignore.      */  
int          G_MaxValueId; /* Max BitIdent found in the '.values'      */  
                  /* section of the fsm file.      */  
int          G_ListLhsVar; /* List of couples :      */  
                  /* (Id1, Id2) where 'Id1' is the Id */  
                  /* of the output/inout/breakpoint */  
                  /* function 'Id2' value of the id */  
                  /* Function.      */  
BLIST       G_ListStateVar; /* List of couples :      */  
                  /* (Id1, Id2) where 'Id1' is the Id */  
                  /* of the output/inout/breakpoint */  
                  /* function 'Id2' value of the id */  
                  /* Function.      */  
BLIST       G_ListInstances; /* List of triplets: (instance_name, */  
                  /* GnlName, File)      */  
  
static int  G_HierarchicalMode; /* 0 -> '-hierarchical' option      */
```

gnl fsm.y

```
/* not used, 1 -> option used. */  
  
static BLIST      G_ListPorts;  
static BLIST      G_ListLoc;  
static char *NewStr;  
  
/*-----*/  
int      yyparse          (void);  
int      yylex           (void);  
  
int    LineNumberReadFsm;  
int    FsmVersion;  
  
/*-----*/  
/* DEFINE */  
/*-----*/  
#define HASH_LIST_SIZE_FSM_VAR      5000  
  
#define TRANSLATE_V2_V3      1  
#define MASTER_CLOCK_EVENT "mc|'event"     /* also defined in type.h */  
#define TICKEVENT           "'event"       /* also defined in fsm.h */  
  
}  
%union {  
    int      integ;  
    char *   string;  
}  
  
%start file  
%token TNULL  
%token FSM LIBRARY COLON OPENPAR CLOSEPAR COMMA DASH SLASH  
%token SIGNATURE PARAMVALUE RETURNRANGE OPTIONS  
%token AXIOMS ASSERTSCLOCKS DATE END EOF FUNCTIONS  
%token INOUTBUSES INPUTS OUTBUSES  
%token PRAGMA STATEVARS VALID INIT INSTANCES  
%token BBINPUTS BBINOUTS BBOUTPUTS BBGENERICS  
%token GLOBALS DONTCARES BREAKPOINTS  
%token VALUES REFMODL DERMODL NAMEMATCH  
%token FILENAMES DEADCODEDETECT ALTNAMES LOC NOT TYPE ENUMERATIONS  
%token ENUM_ENCODING HISTORY ORIGINLANG  
  
%token <integ>  INTEGER TICKNUM FSMVERSION DCD_REGION  
%token <string> STRING BITIDENT ID RANGE TICKEV RECORDFIELD  
%token <string> ENUMERATE KINTEGER HPOINT CHARACT PREFIX  
  
%type <string> BitIdent SimpleBitIdent filename FsmName  
%type <string> HBitIdents HBitIdent Id Attribut strings  
%type <string> typenum enum_encoding enumlit  
%type <string> id_records id_record  
  
%%  
file   :  

```

gnl fsm.y

```
{  
    LineNumberReadFsm = 1;  
    G_MaxValueId = 0;  
    G_NbBddNodes = 0;  
    if (GnlCreate (NULL, &G_CurrentGnl))  
        return (GNL_MEMORY_FULL);  
    if (BListAddElt (G_ListOfGnls, (int)G_CurrentGnl))  
        return (GNL_MEMORY_FULL);  
    if (BListCreateWithSize (HASH_LIST_SIZE_FSM_VAR,  
                           &G_HashListFsmVar))  
        return (GNL_MEMORY_FULL);  
    BSize (G_HashListFsmVar) = HASH_LIST_SIZE_FSM_VAR;  
  
    if (BListCreateWithSize (1, &G_ListFuncIdDontCare))  
        return (GNL_MEMORY_FULL);  
  
    if (BListCreateWithSize (1, &G_ListLhsVar))  
        return (GNL_MEMORY_FULL);  
  
    if (BListCreateWithSize (1, &G_ListStateVar))  
        return (GNL_MEMORY_FULL);  
    SetGnlListStateVar (G_CurrentGnl, G_ListStateVar);  
  
    if (BListCreate (&G_ListPorts))  
        return (GNL_MEMORY_FULL);  
    SetGnlListPorts (G_CurrentGnl, G_ListPorts);  
  
    if (BListCreate (&G_ListLoc))  
        return (GNL_MEMORY_FULL);  
  
    if (BListCreate (&G_ListInstances))  
        return (GNL_MEMORY_FULL);  
    G_HierarchicalMode = 0;  
}  
fsmdecl  
Functions  
{  
    BListQuickDelete (&G_ListLoc);  
}  
;  
  
fsmdecl : Header  
{  
}  
Declarations  
{  
}  
Values  
{  
}  
Axioms Asserts Valid  
{  
}  
;  
  
/* The .fsmversion and the .date are mandatory at the beginning */  
/* .date has the form:      YYYY-MMM-DD HH:mm:ss.hh      */
```

gnl fsm.y

```
Header   : FSMVERSION
          {
          }
Controles
;

Controles : Controles Controle
          |
          ;
Controle  : Signature
          | ParamValue
          | ReturnRange
          | Pragma
          | DATE STRING
          { }
          | FSM FsmName
          { }
          | Library
          | OriginLang
          | HISTORY histories
;

Functions : Funs END
          { YYACCEPT; }
          ;
histories : histories STRING
          {
          }
          |
          ;
/*
 * The Date is optional for non regression purposes */
/* It has the form:      YYYY-MMM-DD HH:mm:ss.hh      */

/*
 * Pragma to store needed informations. It is optional.
 * Trace options of fsmc.
*/
Pragma   : PRAGMA options
          ;
options  : OPTIONS string_options
          ;
string_options : string_options STRING
          {
              if (!strcmp ($2, "-hierarchical"))
                  G_HierarchicalMode = 1;
          }
          |
          ;
strings  : strings STRING
```

gnlfsm.y

```
{  
    char *st = realloc ($1, strlen ($1) + strlen ($2) + 1);  
    $$ = strcat (st, $2);  
    free ($2);  
}  
| STRING  
{ $$ = $1; }  
;  
  
/*  
The signature in the case of a function or a procedure.  
*/  
  
Signature : SIGNATURE strings  
{ }  
;  
/*  
*/  
ParamValue : PARAMVALUE strings  
{ }  
;  
/*  
Function return value.  
*/  
ReturnRange : RETURNRANGE strings  
{ }  
;  
  
/* The name of the FSM is mandatory */  
  
FsmName : FsmName DASH BitIdent  
{  
    if (GnlStrAppendStrCopy ($1, "-", &G_NewName))  
        return (GNL_MEMORY_FULL);  
    $$ = G_NewName;  
    if (GnlStrAppendStrCopy ($$, $3, &G_NewName))  
        return (GNL_MEMORY_FULL);  
    $$ = G_NewName;  
    SetGnlName (G_CurrentGnl, $$);  
}  
| BitIdent  
{ $$ = $1; }  
;  
  
/* The name of the logical LIBRARY */  
Library : LIBRARY STRING  
{ }  
;  
  
/* The original language among {VHDL, Verilog} */  
  
OriginLang : ORIGINLANG ID  
{ }  
  
/* The clock definition is optional. If no clock is defined we suppose */  
/* that the description is purely combinational. */
```

```

clocks : clocks clock
|
;
clock : BitIdent COLON RANGE Attribut INTEGER typenum
{
}
|
BitIdent COLON Attribut INTEGER typenum
{
}
;

/* Variables are grouped in blocks of variables with the same types */
/* Variable blocks are optional. */
/* Note: All variables can have an specified initial value. */

Declarations : Declarations declaration
|
;

declaration : ENUMERATIONS enumerations
CLOCKS      clocks
INPUTS       inputs
OUTBUSES    outbuses
INOUTBUSES inoutbuses
STATEVARS   svars
BBINPUTS    bbinputs
BBINOUTS   bbinouts
BBOUTPUTS  bboutputs
BBGENERICS bbgenerics
BREAKPOINTS breakpoints
DONTCARES   dontcares
GLOBALS     globals
INSTANCES   string_instances
{
    /* If we have instances then we must have the option */
    /* '-hierarchical' present. */
    if (BListSize (G_ListInstances) &&
        !G_HierarchicalMode)
    {
        fprintf (stderr,
"ERROR: Fsmc must be generated with '-hierarchical' option\n");
        return (GNL_BAD_FSM_FOR_SYNTHESIS);
    }

    if (GnlCreateFsmComponents (G_CurrentGnl,
                                G_ListInstances))
        return (GNL_MEMORY_FULL);
    }
    FILENAMES filenames
{
}
| DEADCODEDETECT deadcodedetects
{
}
;

```

```

deadcodedetects : deadcodedetects deadcodedetect
{
}
|
{ }
;
deadcodedetect : INTEGER COLON INTEGER SLASH INTEGER INTEGER DCD_REGION
{
    if (!$6)
    {
        fprintf (stderr,
" WARNING: Dead Code has been detected in file '%s'\n",
GnlGetShortFileName (GnlListSourceFiles (G_CurrentGnl),
$1));
        fprintf (stderr, "           when entering line %d\n", $3);
    }
}
;
filenames : filenames filename
{
}
|
{ }
;
filename : INTEGER STRING
{
    if (GnlStrCopy ($2, &NewStr))
        return (GNL_MEMORY_FULL);
    if (BListAddEl (GnlListSourceFiles (G_CurrentGnl),
(int)NewStr))
        return (GNL_MEMORY_FULL);
}
;
Values : {
    if (GnlCreateAllSignals (G_CurrentGnl))
        return (GNL_MEMORY_FULL);
}
VALUES values
|
;
values : values value
|
;
value : field8 alters
{
    BSize (G_ListLoc) = 0;
}
locats
{
    if (GnlSetLocationInVar (G_Var, G_ListLoc))
        return (GNL_MEMORY_FULL);
}
|
field6 alters
{
}

```

```

        BSize (G_ListLoc) = 0;
    }
locats
{
    if (GnlSetLocationInVar (G_Var, G_ListLoc))
        return (GNL_MEMORY_FULL);
}
| field5 alters
{
    BSize (G_ListLoc) = 0;
}
locats
{
    if (GnlSetLocationInVar (G_Var, G_ListLoc))
        return (GNL_MEMORY_FULL);
}
| field4 alters
{
    BSize (G_ListLoc) = 0;
}
locats
{
    if (GnlSetLocationInVar (G_Var, G_ListLoc))
        return (GNL_MEMORY_FULL);
}
| field3 alters
{
    BSize (G_ListLoc) = 0;
}
locats
{
    if (GnlSetLocationInVar (G_Var, G_ListLoc))
        return (GNL_MEMORY_FULL);
}
;

/* name boolvar boolZ value driving discon init initZ [.loc]  (Out/Inout)
 */
/* name boolvar boolZ value driving discon init initZ [.loc]  (BreakPoint)
 */
field8 : BitIdent INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                    (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    /* Storing the FSM index of the variable on the Id field.      */
    SetGnlVarId (G_Var, $2);
    if ($2 > G_MaxValueId)
        G_MaxValueId = $2;

    GnlAddFsmVarInHashTable (G_HashListFsmVar, G_Var);

    /* We add the triplet (name, value, drive_value) in the list  */
    /* 'G_ListLhsVar'.                                         */
}

```

gnlfsm.y

```
if (BListAddElt (G_ListLhsVar, (int)G_Var))
    return (GNL_MEMORY_FULL);
if (BListAddElt (G_ListLhsVar, (int)$4))
    return (GNL_MEMORY_FULL);
SetGnlVarDriveFuncId (G_Var, $5);
}

;

field6 : BitIdent INTEGER INTEGER INTEGER INTEGER INTEGER
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    /* Storing the FSM index of the variable on the Id field. */
    SetGnlVarId (G_Var, $2);
    if ($2 > G_MaxValueId)
        G_MaxValueId = $2;

    GnlAddFsmVarInHashTable (G_HashListFsmVar, G_Var);

    SetGnlVarDriveFuncId (G_Var, 1);
}
;

field5 : BitIdent INTEGER INTEGER INTEGER INTEGER
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    /* Storing the FSM index of the variable on the Id field. */
    SetGnlVarId (G_Var, $2);
    if ($2 > G_MaxValueId)
        G_MaxValueId = $2;

    GnlAddFsmVarInHashTable (G_HashListFsmVar, G_Var);

    SetGnlVarDriveFuncId (G_Var, 1);
}
;

/* name boolvar value init           [.altname] [.loc]  (Statevar)
 */
field4 : BitIdent INTEGER INTEGER INTEGER
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    /* Storing the FSM index of the variable on the Id field. */
}
```

gnlfsm.y

```
SetGnlVarId (G_Var, $2);
if ($2 > G_MaxValueId)
    G_MaxValueId = $2;

GnlAddFsmVarInHashTable (G_HashListFsmVar, G_Var);

/* We add the couple (name, value) in the list 'G_ListStateVar'.*/
if (BListAddElt (G_ListStateVar, (int)G_Var))
    return (GNL_MEMORY_FULL);
if (BListAddElt (G_ListStateVar, (int)$3))
    return (GNL_MEMORY_FULL);

SetGnlVarDriveFuncId (G_Var, 1);
}

;

field3 : BitIdent INTEGER INTEGER
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
        (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    /* Storing the FSM index of the variable on the Id field. */
    SetGnlVarId (G_Var, $2);
    if ($2 > G_MaxValueId)
        G_MaxValueId = $2;

    GnlAddFsmVarInHashTable (G_HashListFsmVar, G_Var);

    SetGnlVarDriveFuncId (G_Var, 1);
}
;

locats : locats locat
{
}
|
{ }

;

locat : LOC INTEGER COLON INTEGER
{
    if (BListAddElt (G_ListLoc, (int)$2))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (G_ListLoc, (int)$4))
        return (GNL_MEMORY_FULL);
}
;

alters : alters alter
{
}
|
{ }

;
```

gnlfsm.y

```
alter  : ALTNODE NOT BitIdent
      { }
      | ALTNODE BitIdent
      { }
;

typenum: TYPE BitIdent
      { }
      |
      { }
;

enumerations: enumerations enumeration
      { }
      |
      ;
;

enumeration: BitIdent OPENPAR enumlits CLOSEPAR enum_encoding
      {
      }
      ;
;

enumlits: enumlits COMMA enumlit
      {
      }
      | enumlit
      {
      }
      ;
;

enumlit: STRING
      { }
      ;
;

enum_encoding: ENUM_ENCODING STRING
      { }
      |
      { }
      ;
;

/* Declaration of Inputs */
inputs : inputs input
      |
      ;
;

input : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_INPUT);
        if (BListAddElt (G_ListPorts, (int)G_Var))
;
```

```

        return (GNL_MEMORY_FULL);
SetGnlVarLsb (G_Var, $5-1);
SetGnlVarMsb (G_Var, 0);
}
else /* should be "enumerate" */ {
{
if ($5 != 1)
{
fprintf (stderr,
        " WARNING: <%s> is a complex type signal\n",
        $1);
}
else
{
if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                (G_CurrentGnl, $1, &G_Var)))
{
if (G_GnlStatus != GNL_VAR_EXISTS)
    return (G_GnlStatus);
}
SetGnlVarDir (G_Var, GNL_VAR_INPUT);
if (BListAddElt (G_ListPorts, (int)G_Var))
    return (GNL_MEMORY_FULL);

GnlUpdateVarBoundsWithRange (G_Var, $3);
}
}
}
BitIdent COLON Attribut INTEGER typenum
{
if (!strcmp ($3, "Integer"))
{
if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                (G_CurrentGnl, $1, &G_Var)))
{
if (G_GnlStatus != GNL_VAR_EXISTS)
    return (G_GnlStatus);
}
SetGnlVarDir (G_Var, GNL_VAR_INPUT);
if (BListAddElt (G_ListPorts, (int)G_Var))
    return (GNL_MEMORY_FULL);
SetGnlVarLsb (G_Var, $4-1);
SetGnlVarMsb (G_Var, 0);
}
else /* should be "enumerate" */
{
if ($4 != 1)
{
if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                (G_CurrentGnl, $1, &G_Var)))
{
if (G_GnlStatus != GNL_VAR_EXISTS)
    return (G_GnlStatus);
}
SetGnlVarDir (G_Var, GNL_VAR_INPUT);
if (BListAddElt (G_ListPorts, (int)G_Var))
    return (GNL_MEMORY_FULL);
}
}
}

```

```

        SetGnlVarMsb (G_Var, $4-1);
        SetGnlVarLsb (G_Var, 0);

    }
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_INPUT);
    if (BListAddElt (G_ListPorts, (int)G_Var))
        return (GNL_MEMORY_FULL);
}
}
}

/* Declaration of Outbuses */
outbuses : outbuses outbus
|
;
outbus : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_OUTPUT);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
        if (BListAddElt (G_ListPorts, (int)G_Var))
            return (GNL_MEMORY_FULL);
    }
    else /* should be "enumerate" */ 
    {
        if ($5 != 1)
        {
            fprintf (stderr,
                  " WARNING: <%s> is a complex type signal\n",
                  $1);
        }
    else
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
    }
}
}

```

```

        SetGnlVarDir (G_Var, GNL_VAR_OUTPUT);
        if (BListAddElt (G_ListPorts, (int)G_Var))
            return (GNL_MEMORY_FULL);

        GnlUpdateVarBoundsWithRange (G_Var, $3);
    }
}

| BitIdent COLON Attribut INTEGER typenum
{
if (!strcmp ($3, "Integer"))
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                      (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_OUTPUT);
    SetGnlVarLsb (G_Var, $4-1);
    SetGnlVarMsb (G_Var, 0);
    if (BListAddElt (G_ListPorts, (int)G_Var))
        return (GNL_MEMORY_FULL);
    }
else /* should be "enumerate" */                    */
{
    if ($4 != 1)
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                      (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_OUTPUT);
        if (BListAddElt (G_ListPorts, (int)G_Var))
            return (GNL_MEMORY_FULL);

        SetGnlVarMsb (G_Var, $4-1);
        SetGnlVarLsb (G_Var, 0);
    }
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                      (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_OUTPUT);
    if (BListAddElt (G_ListPorts, (int)G_Var))
        return (GNL_MEMORY_FULL);
}
}
}
;
```

gnl fsm.y

```
/* Declaration of Inoutbuses */
inoutbuses : inoutbuses inoutbus
|
;

inoutbus : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_INOUT);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
        if (BListAddElt (G_ListPorts, (int)G_Var))
            return (GNL_MEMORY_FULL);
    }
    else      /* should be "enumerate" */           */
    {
        if ($5 != 1)
        {
            fprintf (stderr,
                      " WARNING: <%s> is a complex type signal\n",
                      $1);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_INOUT);
            if (BListAddElt (G_ListPorts, (int)G_Var))
                return (GNL_MEMORY_FULL);

            GnlUpdateVarBoundsWithRange (G_Var, $3);
        }
    }
}
| BitIdent COLON Attribut INTEGER typenum
{
    if (!strcmp ($3, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_INOUT);
        SetGnlVarLsb (G_Var, $4-1);
```

```

        SetGnlVarMsb (G_Var, 0);
        if (BListAddElt (G_ListPorts, (int)G_Var))
            return (GNL_MEMORY_FULL);
    }
    else /* should be "enumerate" */ {
        if ($4 != 1)
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_INOUT);
            if (BListAddElt (G_ListPorts, (int)G_Var))
                return (GNL_MEMORY_FULL);

            SetGnlVarMsb (G_Var, $4-1);
            SetGnlVarLsb (G_Var, 0);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_INOUT);
            if (BListAddElt (G_ListPorts, (int)G_Var))
                return (GNL_MEMORY_FULL);
            }
        }
    ;
/* Declaration of State Variables */
svars : svarts svar
|
;
svar : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_STATE_VAR);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ {
}

```

```

if ($5 != 1)
{
    fprintf (stderr,
              " WARNING: <%s> is a complex type signal\n",
              $1);
}
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                     (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_STATE_VAR);

    GnlUpdateVarBoundsWithRange (G_Var, $3);
}
}

| BitIdent COLON Attribut INTEGER typenum
{
    if (!strcmp ($3, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                         (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_STATE_VAR);
        SetGnlVarLsb (G_Var, $4-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ 
    {
        if ($4 != 1)
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                             (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_STATE_VAR);

            SetGnlVarMsb (G_Var, $4-1);
            SetGnlVarLsb (G_Var, 0);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                             (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
        }
    }
}

```

```

        SetGnlVarDir (G_Var, GNL_STATE_VAR);
    }
}
;

/* Declaration of BlackBox Inputs */
bbinputs : bbinputs bbinput
|
;
bbinput : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ *
    {
        if ($5 != 1)
        {
            fprintf (stderr,
                     " WARNING: <%s> is a complex type signal\n",
                     $1);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);

            GnlUpdateVarBoundsWithRange (G_Var, $3);
        }
    }
}
BitIdent COLON Attribut INTEGER typenum
{
    if (!strcmp ($3, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
    }
}
;
```

```

        SetGnlVarLsb (G_Var, $4-1);
        SetGnlVarMsb (G_Var, 0);
    }
else /* should be "enumerate" */                                */
{
    if ($4 != 1)
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);

        SetGnlVarMsb (G_Var, $4-1);
        SetGnlVarLsb (G_Var, 0);
    }
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
}
}
}

/* Declaration of BlackBox Inouts */
bbinouts : bbinouts bbinout
|
;
bbinout : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX_INOUT);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
else /* should be "enumerate" */                                */
{
    if ($5 != 1)
    {
        fprintf (stderr,
                  " WARNING: <%s> is a complex type signal\n",
                  $1);
    }
}
}

```

```

        }
    else
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX_INOUT);

        GnlUpdateVarBoundsWithRange (G_Var, $3);
    }
}
| BitIdent COLON Attribut INTEGER typenum
{
if (!strcmp ($3, "Integer"))
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX_INOUT);
    SetGnlVarLsb (G_Var, $4-1);
    SetGnlVarMsb (G_Var, 0);
}
else /* should be "enumerate" */ */
{
    if ($4 != 1)
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX_INOUT);

        SetGnlVarMsb (G_Var, $4-1);
        SetGnlVarLsb (G_Var, 0);
    }
}
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX_INOUT);
}
}
;

```

```

/* Declaration of BlackBox Outputs */
bboutputs : bboutputs bboutput
|
|
bboutput : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ /* */
    {
        if ($5 != 1)
        {
            fprintf (stderr,
                      " WARNING: <%s> is a complex type signal\n",
                      $1);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);

            GnlUpdateVarBoundsWithRange (G_Var, $3);
        }
    }
}
| BitIdent COLON Attribut INTEGER typenum
{
    if (!strcmp ($3, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
        SetGnlVarLsb (G_Var, $4-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ /* */
    {

```

gnl fsm.y

```
if ($4 != 1)
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                      (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);

    SetGnlVarMsb (G_Var, $4-1);
    SetGnlVarLsb (G_Var, 0);
}
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                      (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_FSM_BLACK_BOX);
}
}

/* Declaration of BlackBox Generics */
bbgenerics : bbgeneric
|
;
bbgeneric : BitIdent COLON RANGE Attribut INTEGER typenum
|
|
| BitIdent COLON Attribut INTEGER typenum
|
|
;

/* Declaration of Breakpoints */
breakpoints : breakpoints breakpoint
|
;
breakpoint : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                          (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_LOCAL);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
}
```

```

else      /* should be "enumerate" */           */
{
    if ($5 != 1)
    {
        fprintf (stderr,
                  " WARNING: <%s> is a complex type signal\n",
                  $1);
    }
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_LOCAL);

    GnlUpdateVarBoundsWithRange (G_Var, $3);
}
}
| BitIdent COLON Attribut INTEGER typenum
{
if (!strcmp ($3, "Integer"))
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)
            return (G_GnlStatus);
    }
    SetGnlVarDir (G_Var, GNL_VAR_LOCAL);
    SetGnlVarLsb (G_Var, $4-1);
    SetGnlVarMsb (G_Var, 0);
}
else /* should be "enumerate" */           */
{
    if ($4 != 1)
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_LOCAL);

        SetGnlVarMsb (G_Var, $4-1);
        SetGnlVarLsb (G_Var, 0);
    }
else
{
    if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
          (G_CurrentGnl, $1, &G_Var)))
    {
        if (G_GnlStatus != GNL_VAR_EXISTS)

```

```

                    return (G_GnlStatus);
                }
                SetGnlVarDir (G_Var, GNL_VAR_LOCAL);
            }
        }
    ;
/* Declaration of Globals */
globals : globals global
|
;

global : BitIdent COLON RANGE Attribut INTEGER typenum
{
    fprintf (stderr,
              " WARNING: global variable <%s> is ignored\n",
              $1);
}
| BitIdent COLON Attribut INTEGER typenum
{
    fprintf (stderr,
              " WARNING: global variable <%s> is ignored\n",
              $1);
}
;
/* Declaration of Dont-cares */
dontcares : dontcares dontcare
|
;
dontcare : BitIdent COLON RANGE Attribut INTEGER typenum
{
    if (!strcmp ($4, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_LOCAL_DONTCARE);
        SetGnlVarLsb (G_Var, $5-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ 
    {
        if ($5 != 1)
        {
            fprintf (stderr,
                      " WARNING: <%s> is a complex type signal\n",
                      $1);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)

```

```

                return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_LOCAL_DONTCARE);

            GnlUpdateVarBoundsWithRange (G_Var, $3);
        }
    }

    | BitIdent COLON Attribut INTEGER typenum
{
    if (!strcmp ($3, "Integer"))
    {
        if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
              (G_CurrentGnl, $1, &G_Var)))
        {
            if (G_GnlStatus != GNL_VAR_EXISTS)
                return (G_GnlStatus);
        }
        SetGnlVarDir (G_Var, GNL_VAR_LOCAL_DONTCARE);
        SetGnlVarLsb (G_Var, $4-1);
        SetGnlVarMsb (G_Var, 0);
    }
    else /* should be "enumerate" */ 
    {
        if ($4 != 1)
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_LOCAL_DONTCARE);

            SetGnlVarMsb (G_Var, $4-1);
            SetGnlVarLsb (G_Var, 0);
        }
        else
        {
            if ((G_GnlStatus = GnlVarCreateAndAddInHashTable
                  (G_CurrentGnl, $1, &G_Var)))
            {
                if (G_GnlStatus != GNL_VAR_EXISTS)
                    return (G_GnlStatus);
            }
            SetGnlVarDir (G_Var, GNL_VAR_LOCAL_DONTCARE);
        }
    }
}

string_instances : string_instances STRING
{
    if (BListAddElt (G_ListInstances, (int)$2))
        return (GNL_MEMORY_FULL);
}
|
;
```

```

/* Axioms are optional. When given they are Boolean relations that */
/* hold statically between the variables of the machine. */

Axioms      : AXIOMS axioms
    {
    }
|
;

axioms      : axioms axiom
    {
    }
|
{
}
;

/* An axiom is made of an identifier, a Boolean expression represented */
/* by its typed decision graph, and an optional string. */

axiom : ID TICKNUM INTEGER strings
{
    /* We store the axioms Id in a global list                      */
    if (BListAddElt (G_ListFuncIdDontCare, $3))                  */
        return (GNL_MEMORY_FULL);
}
;
;

/* Asserts are optional. When given they are Boolean relations that */
/* must be checked about the machine. */

Asserts     : ASSERTS asserts
    {
    }
|
;

asserts     : asserts assert
    {
    }
|
{
}
;

/* An assert is made of an identifier, a Boolean expression represented */
/* by its typed decision graph, and an optional string. */

assert      : ID TICKNUM INTEGER strings
{
    /* We store the assert Id in a global list                      */
    if (BListAddElt (G_ListFuncIdDontCare, $3))                  */
        return (GNL_MEMORY_FULL);

    GnlWarnWithAssert ($4);
}
;
;
```

```

/* Valid is optional. When given, it is the characteristic function of */
/* the reachable or valid states of the machine. This function is */
/* represented by its typed decision graph. */

Valid : VALID INTEGER
    {
    }
|
;

/* The "funs" part of the description is the set of vertices that form */
/* the typed decision graphs of all the Boolean functions used by all */
/* the boolean functions of the machine. */

Funs : FUNCTIONS
funs
{
    /* Once we generated the functions related to '.functions'      */
    /* we generate the functions related to '.values'.           */
    if (GnlCreateFsmFunctionFromValues (G_CurrentGnl,
                                         G_HashListFsmVar, G_ListLhsVar,
                                         G_ListStateVar, G_MaxValueId))
        return (GNL_MEMORY_FULL);
    }
;
;

funs : funs fun
{
    G_NbBddNodes++;
}
|
;

/* Each vertex is defined by a unique number > 2, the order of the variable
*/
/* at the root of the vertex, and the numbers of the left and right */
/* vertices pointed to by the current vertex. Note that there are two */
/* predefined vertex numbers 0 and 1 that represent the boolean values */
/* false and true respectively. */

fun : INTEGER INTEGER INTEGER INTEGER
{
    if (GnlCreateFsmFunction (G_CurrentGnl, G_HashListFsmVar,
                             G_ListFuncIdDontCare, G_MaxValueId,
                             $1, $2, $3, $4))
        return (GNL_MEMORY_FULL);
}
;

Attribut : ENUMERATE | KINTEGER ;

BitIdent : HBitIdents SimpleBitIdent TICKEV
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
}
;
```

gnlfsm.y

```
    $$ = G_NewName;
    if (GnlStrAppendStrCopy ($$, $3, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
| HBitIdents SimpleBitIdent
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
;
/* Hierarchical bitident: "abc.def." */
HBitIdents : HBitIdents HBitIdent
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
|
{ $$ = strdup (""); }
;

HBitIdent : SimpleBitIdent HPOINT
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
;

SimpleBitIdent : PREFIX SimpleBitIdent
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
| id_records Id
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
| Id
{ $$ = $1; }
;

id_records : id_records id_record
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
| id_record
{ $$ = $1; }
;
```

gnl fsm.y

```
id_record : Id RECORDFIELD
{
    if (GnlStrAppendStrCopy ($1, $2, &G_NewName))
        return (GNL_MEMORY_FULL);
    $$ = G_NewName;
}
;

Id      : ID
{
    $$ = $1;
}
| BITIDENT
{
    $$ = $1;
}
;

%%
```

```
/*-----*/
```

gnlfsmread.c

```
/*-----*/
/*
 *      File:          gnlfsmread.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "bbdd.h"
#include "gnloption.h"

#include "bbdd.e"

/*-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV    G_GnlEnv;
extern int   G_NbBddNodes;

/*-----*/
/* DEFINE */
/*-----*/
#define GNL_MASTER_CLOCK_EVENT      "mc@'event"

/*-----*/
/* GLOBAL VARIABLES */
/*-----*/
extern GNL  G_CurrentGnl;           /* from gnlread.c */           */
extern BLIST  G_ListOfGnls;         /* from gnlread.c */          */
                                         */
extern int  IntIdentical ();

/*-----*/
/* yyerror */
/*-----*/
void yyerror (char* s)
{
    fprintf (stderr, " ERROR: during FSM parsing.\n");
}

/*-----*/
/* yywrap */
/*-----*/
int yywrap (void)
{
    return (1);
}
```

gnlfsmread.c

```
}

/*-----*/
/* GnlGetShortFileName */
/*-----*/
char *GnlGetShortFileName (ListFileNames, Index)
    BLIST      ListFileNames;
    int         Index;
{
    char          *FileName;
    int           L;
    char          *c;
    int           i;

    FileName = (char*)BListElt (ListFileNames, Index-1);
    L = strlen (FileName);
    c = FileName;
    i = L-1;
    while ((c[i] != '/') && (i >= 0)) i--;
    if (i == 0)
        return (FileName);

    i++;
    c = c+i;

    return (c);
}

/*-----*/
/* GnlWarnWithAssert */
/*-----*/
/* This procedure analyzes the Assert message found in the .fsm generated*/
/* by FSMC and prints out eventually a message in MAPIT. */
/*-----*/
void GnlWarnWithAssert (String)
    char          *String;
{
    int           L;
    int           i;
    char          *c;

    /* an axiom message is composed of 3 '=' ex: */
    /* ORIGIN=FSMC_Message SEVERITY=ERROR REPORT=The different drivers */
    /* of dataout (9) have incompatible values */
    L = strlen (String);
    i = 0;
    c = String;
    while ((c[i] != '=') && (i<L)) i++;

    i++;
    if (i >= L)
        return;

    while ((c[i] != '=') && (i<L)) i++;
```

gnlfsmread.c

```
i++;
if (i >= L)
    return;

while ((c[i] != '=') && (i<L)) i++;

i++;
if (i >= L)
    return;

if (c[i] == '\n')
    return;

fprintf (stderr, " WARNING: ");
while ((c[i] != '\n') && (i < L))
{
    fprintf (stderr, "%c", c[i]);
    i++;
}

fprintf (stderr, "\n");
}

/*
 * -----
 * GnlAddFsmVarInHashTable
 * -----
 * This procedure adds the Var 'Var' according to its Id value which
 * corresponds to its fsm index in the .fsm file. If the Var is already
 * in the Hash table then 'GNL_VAR_EXISTS' is returned. If a memory
 * problem occurs then 'GNL_MEMORY_FULL' is returned and 'GNL_OK' if
 * everything is fine.
 * -----
 * GNL_STATUS GnlAddFsmVarInHashTable (HashListFsmVar, Var)
 *      BLIST      HashListFsmVar;
 *      GNL_VAR    Var;
 {
     int          Key;
     int          i;
     BLIST        Bucket;
     GNL_VAR     VarI;

     Key = (int)GnlVarId (Var) % BListSize (HashListFsmVar);
     Bucket = (BLIST)BListElt (HashListFsmVar, Key);
     if (!Bucket)
     {
         if (BListCreateWithSize (1, &Bucket))
             return (GNL_MEMORY_FULL);
         BListElt (HashListFsmVar, Key) = (int)Bucket;
     }

     /* Does the var already exist ?
     for (i=0; i<BListSize (Bucket); i++)
     {

```

gnl fsmread.c

```

VarI = (GNL_VAR)BListElt (Bucket, i);
if (GnlVarId (Var) == GnlVarId (VarI))
    return (GNL_VAR_EXISTS);
}

if (BListAddElt (Bucket, (int)Var))
    return (GNL_MEMORY_FULL);

/*
fprintf (stderr, " ADDING Var <%s> with Id [%d]\n",
        GnlVarName (Var), (int)GnlVarId (Var));
*/
return (GNL_OK);
}

/*-----
/* GnlGetFsmVarFromId
/*-----
/* This procedure updates the fields 'Msb' and 'Lsb' of the GNL_VAR
/* 'Var' by extracting the values from the string 'Range'.
/*-----*/
void GnlUpdateVarBoundsWithRange (Var, Range)
    GNL_VAR Var;
    char *Range;
{
    int i;
    char MsbString[128]; /* we store an integer in this object
    */
    char LsbString[128]; /* we store an integer in this object
    */
    int Index;
    int Msb;
    int Lsb;

    i=1;
    while (i<strlen (Range))
    {
        /* If the end of the integer we break.
        if (Range[i] == ' ')
            break;
        MsbString[i-1]=Range[i];
        i++;
    }
    MsbString[i-1] = '\0';

    i++;
    while ((i<strlen (Range)) && (Range[i] != ' '))
        i++;
    i++;
    Index = i;
    while (i<strlen (Range))
    {
        /* If the end of the integer we break.
        if (Range[i] == ')')
            break;
        */
    }
}

```

gnlfsmread.c

```
        LsbString[i-Index]=Range[i];
        i++;
    }
    LsbString[i-Index] = '\0';

    Msb = atoi (MsbString);
    Lsb = atoi (LsbString);

    SetGnlVarMsb (Var, Msb);
    SetGnlVarLsb (Var, Lsb);
}

/*-----
/* GnlGetFsmVarFromId
/*-----
/* This procedure returns the GNL_VAR having the fsm id 'Id'. If there
/* is no such var then 'Var' is NULL.
/*-----
void GnlGetFsmVarFromId (HashListFsmVar, Id, Var)
    BLIST      HashListFsmVar;
    int        Id;
    GNL_VAR    *Var;
{
    int        Key;
    int        i;
    BLIST      Bucket;
    GNL_VAR    Vari;

    *Var = NULL;

    /* case of the 0 and 1 values.
    if ((Id == 0) || (Id == 1))
        return;

    Key = Id % BListSize (HashListFsmVar);
    Bucket = (BLIST)BListElt (HashListFsmVar, Key);
    if (!Bucket)
        return;

    /* Does the var already exist ?
    for (i=0; i<BListSize (Bucket); i++)
    {
        Vari = (GNL_VAR)BListElt (Bucket, i);
        if ((int)GnlVarId (Vari) == Id)
        {
            *Var = Vari;
            return;
        }
    }
}

/*-----
/* GnlCreateNewFsmVar
/*-----
GNL_STATUS GnlCreateNewFsmVar (Gnl, HashListFsmVar, MaxValueId,
```

gnlfsmread.c

```
                                Id, NewVar)
GNL          Gnl;
BLIST        HashListFsmVar;
int          MaxValueId;
int          Id;
GNL_VAR     *NewVar;
{
    GNL_STATUS  GnlStatus;
    char        *NewName;

if (GnlStrAppendIntCopy ("\\f", Id, &NewName))
    return (GNL_MEMORY_FULL);

if (GnlVarCreateAndAddInHashTable (Gnl, NewName, NewVar))
    return (GNL_MEMORY_FULL);

SetGnlVarDir (*NewVar, GNL_VAR_FSM_FUNCTION);
SetGnlVarId (*NewVar, Id+MaxValueId);

/* If != 0 then it is a GNL_MEMORY_FULL
if (GnlAddFsmVarInHashTable (HashListFsmVar, *NewVar))
    return (GNL_MEMORY_FULL); */

return (GNL_OK);
}

/*
 * GnlCreateFsmGnlNode
 */
GNL_STATUS GnlCreateFsmGnlNode (Gnl, VarTop, VarLeft, VarRight,
                                LeftId, RightId, LeftPol, RightPol,
                                NewNode)
{
    GNL          Gnl;
    GNL_VAR     VarTop;
    GNL_VAR     VarLeft;
    GNL_VAR     VarRight;
    int          LeftId;
    int          RightId;
    int          LeftPol;
    int          RightPol;
    GNL_NODE   *NewNode;
{
    GNL_NODE   TempNode;
    GNL_NODE   TempNode1;
    GNL_NODE   TempNode2;
    GNL_NODE   TempNode3;
    GNL_NODE   TempNode4;
    GNL_NODE   TempNode5;
    GNL_NODE   TempNode6;
    GNL_NODE   TempNode7;
    BLIST      NewList;

/* If case of a simple bdd node
/* Bdd of the form [x (0) (1)] --> x
*/ */
*/
```

gnlfsmread.c

```
/* Bdd of the form [x (1) (0)] --> !x */  
/* forms [x (0) (0)] and [x (1) (1)] cannot happened. */  
if (!VarLeft && !VarRight)  
{  
    /* either (LeftId, RightId) == (0, 1) or (1, 0) */  
    if ((LeftId == 0) && (RightId == 1))  
    {  
        /* It is in normal form. */  
        if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (*NewNode, (BLIST)VarTop);  
    }  
    else  
    {  
        /* It is in complemented form. */  
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (TempNode, (BLIST)VarTop);  
        if (GnlCreateNodeNot (Gnl, TempNode, NewNode))  
            return (GNL_MEMORY_FULL);  
    }  
    return (GNL_OK);  
}  
/* Bdd of the form [x (1) y] --> !x+y */  
/* Bdd of the form [x (0) y] --> x.y */  
if (!VarLeft)  
{  
    if (LeftId == 1)  
    {  
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (TempNode1, (BLIST)VarTop);  
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode2))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (TempNode2, (BLIST)VarRight);  
        if (RightPol)  
        {  
            if (GnlCreateNodeNot (Gnl, TempNode2, &TempNode7))  
                return (GNL_MEMORY_FULL);  
            TempNode2 = TempNode7;  
        }  
  
        if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode3))  
            return (GNL_MEMORY_FULL);  
        if (GnlCreateNode (Gnl, GNL_OR, NewNode))  
            return (GNL_MEMORY_FULL);  
        if (BListCreateWithSize (2, &NewList))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (NewList, (int)TempNode3))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (NewList, (int)TempNode2))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (*NewNode, NewList);  
    }  
    else  
    {  
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))  
            return (GNL_MEMORY_FULL);  
        if (GnlCreateNode (Gnl, GNL_OR, NewNode))  
            return (GNL_MEMORY_FULL);  
        if (BListCreateWithSize (2, &NewList))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (NewList, (int)TempNode1))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (NewList, (int)TempNode2))  
            return (GNL_MEMORY_FULL);  
        SetGnlNodeSons (*NewNode, NewList);  
    }  
}
```

```

        return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode1, (BLIST)VarTop);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode2, (BLIST)VarRight);
if (RightPol)
{
    if (GnlCreateNodeNot (Gnl, TempNode2, &TempNode7))
        return (GNL_MEMORY_FULL);
    TempNode2 = TempNode7;
}
if (GnlCreateNode (Gnl, GNL_AND, NewNode))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode1))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*NewNode, NewList);
}
return (GNL_OK);
}

/* Bdd of the form [x y (1)] --> x+y */
/* Bdd of the form [x y (0)] --> !x.y */
if (!VarRight)
{
    if (RightId == 0)
    {
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
            return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode1, (BLIST)VarTop);
        if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode2))
            return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode2, (BLIST)VarLeft);
        if (LeftPol)
        {
            if (GnlCreateNodeNot (Gnl, TempNode2, &TempNode7))
                return (GNL_MEMORY_FULL);
            TempNode2 = TempNode7;
        }
        if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode3))
            return (GNL_MEMORY_FULL);
        if (GnlCreateNode (Gnl, GNL_AND, NewNode))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (2, &NewList))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (NewList, (int)TempNode3))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (NewList, (int)TempNode2))
            return (GNL_MEMORY_FULL);
SetGnlNodeSons (*NewNode, NewList);
    }
else
{
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))

```

```

        return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode1, (BLIST)VarTop);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode2, (BLIST)VarLeft);
if (LeftPol)
{
    if (GnlCreateNodeNot (Gnl, TempNode2, &TempNode7))
        return (GNL_MEMORY_FULL);
    TempNode2 = TempNode7;
}
if (GnlCreateNode (Gnl, GNL_OR, NewNode))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode1))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*NewNode, NewList);
}
return (GNL_OK);
}

/* Otherwise this is the general case: [x y z] --> !x.y+x.z */
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode1, (BLIST)VarTop);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode2))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode2, (BLIST)VarTop);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode3))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode3, (BLIST)VarLeft);
if (LeftPol)
{
    if (GnlCreateNodeNot (Gnl, TempNode3, &TempNode7))
        return (GNL_MEMORY_FULL);
    TempNode3 = TempNode7;
}
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode4))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode4, (BLIST)VarRight);
if (RightPol)
{
    if (GnlCreateNodeNot (Gnl, TempNode4, &TempNode7))
        return (GNL_MEMORY_FULL);
    TempNode4 = TempNode7;
}

if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode5))
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (Gnl, GNL_AND, &TempNode6))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode5))

```

gnlfsmread.c

```
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode3))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode6, NewList);

if (GnlCreateNode (Gnl, GNL_AND, &TempNode7))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode2))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode4))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode7, NewList);

if (GnlCreateNode (Gnl, GNL_OR, NewNode))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode6))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode7))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*NewNode, NewList);

    return (GNL_OK);
}

/*-----
/* GnlAddDriveFunction
/*-----*/
GNL_STATUS GnlAddDriveFunction (Gnl, HashListFsmVar, Function,
                                IdDriveFunc, MaxValueId)
{
    GNL             Gnl;
    BLIST           HashListFsmVar;
    GNL_FUNCTION    Function;
    int             IdDriveFunc;
    int             MaxValueId;
{
    int             FuncPol;
    GNL_VAR         VarFunc;
    GNL_NODE        NewNode;
    GNL_NODE        TempNode;

    /* driving function is the tautology. */
    if (IdDriveFunc == 1)
        return (GNL_OK);

    /* driving function is 0 --> always 'Z' */
    if (IdDriveFunc == 0)
    {
        fprintf (stderr, " WARNING: 'Z' signal.\n");
        exit (1);
    }
}
```

gnlfsmread.c

```
FuncPol = 0;
if (IdDriveFunc < 0)
{
    FuncPol = 1;
    IdDriveFunc = -IdDriveFunc;
}

/* Id of the function must be incremented of 'MaxValueId'. */
GnlGetFsmVarFromId (HashListFsmVar, IdDriveFunc+MaxValueId, &VarFunc);

if (!VarFunc)
{
    fprintf (stderr,
            " ERROR: cannot find Id = %d in .values section\n",
            IdDriveFunc);
    exit (1);
}

if (GnlCreateNode (Gnl, GNL_VARIABLE, &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NewNode, (BLIST)VarFunc);
if (FuncPol)
{
    if (GnlCreateNodeNot (Gnl, NewNode, &TempNode))
        return (GNL_MEMORY_FULL);
    NewNode = TempNode;
}

SetGnlFunctionZSet (Function, NewNode);

return (GNL_OK);
}

/*
 * GnlCreateFsmFunctionFromValues
 */
/*
 * This procedure creates the boolean functions for each variables
 * defined in the 'values' part of the FSM file.
 */
GNL_STATUS GnlCreateFsmFunctionFromValues (Gnl, HashListFsmVar,
                                            ListLhsVar, ListStateVar,
                                            MaxValueId)
{
    GNL          Gnl;
    BLIST        HashListFsmVar;
    BLIST        ListLhsVar;
    BLIST        ListStateVar;
    int          MaxValueId;
{
    int          i;
    GNL_VAR     VarLhs;
    GNL_VAR     VarFunc;
    int          IdFunc;
    GNL_NODE    NewNode;
    GNL_FUNCTION NewFunction;
    int          FuncPol;
```

gnlfsmread.c

```
GNL_NODE TempNode;
int IdDriveFunc;

/* 'ListLhsVar' is a list of couples (BitIdent, FunctionId). */
for (i=0; i<BListSize (ListLhsVar)-1; i = i+2)
{
    VarLhs = (GNL_VAR)BListElt (ListLhsVar, i);
    IdFunc = (int)BListElt (ListLhsVar, i+1);
    IdDriveFunc = GnlVarDriveFuncId (VarLhs);

    FuncPol = 0;

    /* Actually function is 0. */
    if (IdFunc == 0)
    {
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, &NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (NewNode, (BLIST) 0);

        if (GnlFunctionCreate (Gnl, VarLhs, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

        SetGnlVarFunction (VarLhs, NewFunction);

        if (BListAddElt (GnlFunctions (Gnl), (int)VarLhs))
            return (GNL_MEMORY_FULL);

        SetGnlFunctionOnSet (NewFunction, NewNode);

        GnlAddDriveFunction (Gnl, HashListFsmVar, NewFunction,
                            IdDriveFunc, MaxValueId);

        continue;
    }

    /* Actually function is 1. */
    if (IdFunc == 1)
    {
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, &NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (NewNode, (BLIST) 1);

        if (GnlFunctionCreate (Gnl, VarLhs, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

        SetGnlVarFunction (VarLhs, NewFunction);

        if (BListAddElt (GnlFunctions (Gnl), (int)VarLhs))
            return (GNL_MEMORY_FULL);

        SetGnlFunctionOnSet (NewFunction, NewNode);

        GnlAddDriveFunction (Gnl, HashListFsmVar, NewFunction,
                            IdDriveFunc, MaxValueId);
    }
}
```

```

        continue;
    }

/* The var is pointed with a complement. */
if (IdFunc < 0)
{
    FuncPol = 1;
    IdFunc = -IdFunc;
}

/* Id of the function must be incremented of 'MaxValueId'. */
GnlGetFsmVarFromId (HashListFsmVar, IdFunc+MaxValueId, &VarFunc);
if (!VarFunc)
{
    fprintf (stderr,
            " ERROR: cannot find Id = %d in .values section\n",
            IdFunc);
    continue;
}

/* We create the new equation and adds it to the list of */
/* equations. */
if (GnlFunctionCreate (Gnl, VarLhs, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

SetGnlVarFunction (VarLhs, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), (int)VarLhs))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (Gnl, GNL_VARIABLE, &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NewNode, (BLIST)VarFunc);
if (FuncPol)
{
    if (GnlCreateNodeNot (Gnl, NewNode, &TempNode))
        return (GNL_MEMORY_FULL);
    NewNode = TempNode;
}

SetGnlFunctionOnSet (NewFunction, NewNode);

GnlAddDriveFunction (Gnl, HashListFsmVar, NewFunction,
                     IdDriveFunc, MaxValueId);
}

/* 'ListStateVar' is a list of couples (BitIdent, FunctionId) */
for (i=0; i<BListSize (ListStateVar)-1; i = i+2)
{
    VarLhs = (GNL_VAR)BListElt (ListStateVar, i);
    IdFunc = (int)BListElt (ListStateVar, i+1);
    FuncPol = 0;

/* Actually no function for this var. */
}

```

gnlfsmread.c

```
if (IdFunc == 0)
    continue;

if (IdFunc < 0)
{
    FuncPol = 1;
    IdFunc = -IdFunc;
}

/* Id of the function must be incremented of 'MaxValueId'. */
GnlGetFsmVarFromId (HashListFsmVar, IdFunc+MaxValueId, &VarFunc);
if (!VarFunc)
{
    fprintf (stderr,
              " ERROR: cannot find Id = %d in .values section\n",
              IdFunc);
    continue;
}

if (GnlFunctionCreate (Gnl, VarLhs, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

SetGnlVarFunction (VarLhs, NewFunction);

if (GnlCreateNode (Gnl, GNL_VARIABLE, &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NewNode, (BLIST)VarFunc);
if (FuncPol)
{
    if (GnlCreateNodeNot (Gnl, NewNode, &TempNode))
        return (GNL_MEMORY_FULL);
    NewNode = TempNode;
}

SetGnlFunctionOnSet (NewFunction, NewNode);

}

return (GNL_OK);
}

/*
 * GnlCreateFsmFunction
 */
/*
 * This procedure creates a new function in the current gnl 'Gnl'. If
 * The index is related to an axiom index then we do not create it
 */
GNL_STATUS GnlCreateFsmFunction (Gnl, HashListFsmVar, ListAxiomsId,
                                 MaxValueId, FuncId, TopId,
                                 LeftId, RightId)
GNL          Gnl;
BLIST      HashListFsmVar;
BLIST      ListAxiomsId;
int       FuncId;
```

gnlfsmread.c

```
int          TopId;
int          LeftId;
int          RightId;
{
    GNL_VAR      OutVar;
    GNL_VAR      VarTop;
    GNL_VAR      VarLeft;
    GNL_VAR      VarRight;
    GNL_FUNCTION NewFunction;
    GNL_NODE     NewNode;
    int          LeftPol;
    int          RightPol;

    /* If the associated function is related to an axiom we do not */
    /* consider it.                                                 */
    if (BListMemberOfList (ListAxiomsId, FuncId, IntIdentical))
        return (GNL_OK);

    /* Create a new var for the new fsm function.                  */
    if (GnlCreateNewFsmVar (Gnl, HashListFsmVar, MaxValueId, FuncId,
                           &OutVar))
        return (GNL_MEMORY_FULL);

    GnlGetFsmVarFromId (HashListFsmVar, TopId, &VarTop);
    if (!VarTop)
    {
        fprintf (stderr, " ERROR: cannot find Id = %d\n", TopId);
        return (GNL_OK);
    }

    /* New fsm var which is not 0 nor 1.                         */
    VarLeft = NULL;
    LeftPol = 0;
    if ((LeftId != 1) && (LeftId != 0))
    {
        if (LeftId < 0)
        {
            LeftPol = 1;
            LeftId = -LeftId;
        }

        GnlGetFsmVarFromId (HashListFsmVar, LeftId+MaxValueId, &VarLeft);
        if (!VarLeft)
        {
            fprintf (stderr, " ERROR: cannot find Id = %d\n", LeftId);
            return (GNL_OK);
        }
    }

    /* New fsm var which is not 0 nor 1.                         */
    VarRight = NULL;
    RightPol = 0;
    if ((RightId != 1) && (RightId != 0))
    {
        if (RightId < 0)
        {
```

gnl fsmread.c

```
RightPol = 1;
RightId = -RightId;
}

GnlGetFsmVarFromId (HashListFsmVar, RightId+MaxValueId, &VarRight);
if (!VarRight)
{
    fprintf (stderr, " ERROR: cannot find Id = %d\n", RightId);
    return (GNL_OK);
}
}

if (GnlFunctionCreate (Gnl, OutVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

SetGnlVarFunction (OutVar, NewFunction);

if (BListAddElt (GnlFunctions(Gnl), (int)OutVar))
    return (GNL_MEMORY_FULL);

if (GnlCreateFsmGnlNode (Gnl, VarTop, VarLeft, VarRight,
                         LeftId, RightId, LeftPol, RightPol, &NewNode))
    return (GNL_MEMORY_FULL);

SetGnlFunctionOnSet (NewFunction, NewNode);

return (GNL_OK);
}

/* -----
/* GnlGetGnlVarFromBddIndex                               */
/* ----- */                                           */
GNL_VAR GnlGetGnlVarFromBddIndex (BddIndex)
    int          BddIndex;
{

    return ((GNL_VAR) BddVarHook ((BDD_VAR) BListElt (GLOB_BDD_WS->Var,
                                         BddIndex)));
}

/* -----
/* GnlBuildBddOnFsmVarInput                            */
/* ----- */                                           */
BDD_STATUS GnlBuildBddOnFsmVarInput (Gnl)
    GNL        Gnl;
{
    int          i;
    int          j;
    GNL_VAR      VarJ;
    BDD_STATUS   Status;
    BDD_VAR      VarBdd;
    BLIST       HashTableNames;
    BLIST       BucketI;

    HashTableNames = GnlHashNames (Gnl);
```

gnlfsmread.c

```
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        if (Status = CreateBddVar (GnlVarName (VarJ), &VarBdd))
            return (Status);

        /* We store the associated Bdd of the GNL_VAR 'VarJ' */
        /* on its Hook field. */
        SetGnlVarHook (VarJ, UseBdd (VarBdd->Bdd));

        /* keeping the reference between the Bdd and the GNL_VAR */
        SetBddVarHook (VarBdd, (int*)VarJ);
    }
}

return (GNL_OK);
}

/*-----
/* GnlConcatListElt
*/
GNL_STATUS GnlConcatListElt (ListFct, Var)
    BLIST    ListFct;
    GNL_VAR  Var;
{
    if (!BListMemberOfList (ListFct, Var, IntIdentical))
    {
        if (BListAddElt (ListFct, (int)Var))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlConcatList
*/
GNL_STATUS GnlConcatList (ListFct, ListFct1)
    BLIST    ListFct;
    BLIST    ListFct1;
{
    int          i;
    GNL_VAR    VarI;

    for (i=0; i<BListSize (ListFct1); i++)
    {
        VarI = (GNL_VAR)BListElt (ListFct1, i);
        if (GnlConcatListElt (ListFct, VarI))
            return (GNL_MEMORY_FULL);
    }
}
```

gnlfsmread.c

```
    return (GNL_OK);
}

/*-----
/* GnlBuildBddFsmExprOnNode
/*-----
/* This procedure buils teh corresponding Bdd of the GNL_NODE associated*/
/* to 'OutVar'. We stop the recursion when the state var is encountered */
/* the second time (Lvl=1). The recursion work only on the variables of */
/* type GNL_VAR_FSM_FUNCTION.                                         */
/*-----*/
BDD_STATUS GnlBuildBddFsmExprOnNode (OutVar, StateVar, Node, Lvl,
                                      Mode, VarMode, ListFct, ListFctMode,
                                      Bdd)
{
    GNL_VAR    OutVar;
    GNL_VAR    StateVar;
    GNL_NODE   Node;
    int         Lvl;
    int         Mode;
    GNL_VAR    VarMode;
    BLIST     ListFct;
    BLIST     ListFctMode;
    BDD        *Bdd;
    {
        int         i;
        GNL_VAR    Var;
        BLIST     Sons;
        GNL_FUNCTION Function;
        GNL_NODE   NodeFunction;
        GNL_NODE   SonI;
        BDD        BddRes;
        BDD        BddTemp;
        BDD        BddI;
        int         NewLvl;
        int         NewMode;

        if (GnlNodeOp (Node) == GNL_CONSTANTE)
        {
            if (GnlNodeSons (Node))
            {
                *Bdd = bdd_one ();
                return (BDD_OK);
            }
            *Bdd = bdd_zero ();
            return (BDD_OK);
        }

        if (GnlNodeOp (Node) == GNL_VARIABLE)
        {
            Var = (GNL_VAR) GnlNodeSons (Node);

            /* No function associated or it is the 'StateVar' or the direction*/
            /* is not 'GNL_VAR_FSM_FUNCTION'.                                         */
            /* If 'ListFct' is NULL we come from the processing of tri-state  */
            /* input and enable functions.                                         */
            if (!GnlVarFunction (Var) || !ListFct)
```

```

{
    *Bdd = UseBdd ((BDD) GnlVarHook (Var));
    if (Mode)
    {
        if (GnlConcatList (ListFct, ListFctMode))
            return (BDD_MEMORY_FULL);
    }
    return (BDD_OK);
}

if ((GnlVarDir (Var) == GNL_VAR_INPUT) ||
    (GnlVarDir (Var) == GNL_VAR_OUTPUT) ||
    (GnlVarDir (Var) == GNL_VAR_INOUT) ||
    (GnlVarDir (Var) == GNL_OUT_STATE_VAR))
{
    *Bdd = UseBdd ((BDD) GnlVarHook (Var));
    if (Mode)
    {
        if (GnlConcatList (ListFct, ListFctMode))
            return (BDD_MEMORY_FULL);
    }
    return (BDD_OK);
}

Function = GnlVarFunction (Var);
NodeFunction = GnlFunctionOnSet (Function);

NewLvl = Lvl;
NewMode = Mode;

if (StateVar == Var)
{
    /* second time we pass thru 'StateVar' in the rec. call */
    if (Lvl == 1)
    {
        if (Mode)
        {
            if (GnlConcatList (ListFct, ListFctMode))
                return (BDD_MEMORY_FULL);
        }
        *Bdd = UseBdd ((BDD) GnlVarHook (Var));
        return (BDD_OK);
    }
    NewLvl = Lvl+1;
}
else if (GnlVarDir (Var) != GNL_VAR_FSM_FUNCTION)
{
    if (GnlVarDir (Var) != GNL_VAR_LOCAL)
    {
        if (Mode)
        {
            if (GnlConcatList (ListFct, ListFctMode))
                return (BDD_MEMORY_FULL);
        }
        *Bdd = UseBdd ((BDD) GnlVarHook (Var));
        return (BDD_OK);
    }
}

```

gnlfsmread.c

```
if (GnlNodeOp (NodeFunction) != GNL_VARIABLE)
{
    if (Mode)
    {
        if (GnlConcatList (ListFct, ListFctMode))
            return (BDD_MEMORY_FULL);
    }
    *Bdd = UseBdd ((BDD) GnlVarHook (Var));
    return (BDD_OK);
}
VarMode = Var;
NewMode = 1;
}
else
{
    if (Mode && (GnlNodeOp (NodeFunction) != GNL_VARIABLE))
    {
        *Bdd = UseBdd ((BDD) GnlVarHook (VarMode));
        return (BDD_OK);
    }
}
if (NewMode)
{
    if (BListAddElt (ListFctMode, (int)Var))
        return (GNL_MEMORY_FULL);
}
else if (ListFct)
{
    if (GnlConcatListElt (ListFct, (int)Var))
        return (GNL_MEMORY_FULL);
}
if (GnlBuildBddFsmExprOnNode (OutVar, StateVar, NodeFunction,
                               NewLvl, NewMode, VarMode,
                               ListFct, ListFctMode, Bdd))
    return (BDD_MEMORY_FULL);

if (NewMode)
{
    BSize (ListFctMode) = BSize (ListFctMode)-1;
}

return (BDD_OK);
}

Sons = GnlNodeSons (Node);
if (GnlNodeOp (Node) == GNL_NOT)
{
    SonI = (GNL_NODE) BListElt (Sons, 0);
    if (GnlBuildBddFsmExprOnNode (OutVar, StateVar, SonI, Lvl,
                                   Mode, VarMode, ListFct, ListFctMode,
                                   &BddRes))
        return (BDD_MEMORY_FULL);
    if (bdd_not (BddRes, Bdd))
        return (BDD_MEMORY_FULL);
    return (BDD_OK);
}
```

gnl fsmread.c

```
}

if (GnlNodeOp (Node) == GNL_AND)
    BddRes = bdd_one ();
else
    /* else GNL_OR. */                                */
    BddRes = bdd_zero ();

for (i=0; i<BListSize (Sons); i++)
{
    SonI = (GNL_NODE)BListElt (Sons, i);
    if (GnlBuildBddFsmExprOnNode (OutVar, StateVar, SonI, Lvl,
                                   Mode, VarMode, ListFct, ListFctMode,
                                   &BddI))
        return (BDD_MEMORY_FULL);

    switch (GnlNodeOp (Node)) {
        case GNL_AND:
            if (bdd_and (BddI, BddRes, &BddTemp))
                return (BDD_MEMORY_FULL);
            break;

        case GNL_OR:
            if (bdd_or (BddI, BddRes, &BddTemp))
                return (BDD_MEMORY_FULL);
            break;

        default:
            fprintf (stderr, "ERROR: unknown operator\n");
            exit (1);
    }

    BddRes = BddTemp;
}

*Bdd = BddRes;

return (BDD_OK);
}

/*-----*/
/* GnlBuildBddFsmExpr */                           */
/*-----*/
BDD_STATUS GnlBuildBddFsmExpr (OutVar, StateVar, ListFct, Bdd)
    GNL_VAR OutVar;
    GNL_VAR StateVar;
    BLIST *ListFct;
    BDD *Bdd;
{
    GNL_FUNCTION Function;
    BLIST ListFctMode;

    Function = GnlVarFunction (OutVar);
```

gnlfsmread.c

```
if (BListCreateWithSize (1, ListFct))
    return (BDD_MEMORY_FULL);

if (BListCreateWithSize (1, &ListFctMode))
    return (BDD_MEMORY_FULL);

if (BListAddElt (*ListFct, OutVar))
    return (BDD_MEMORY_FULL);

if (GnlBuildBddFsmExprOnNode (OutVar, StateVar,
                               GnlFunctionOnSet (Function), 0, 0, NULL,
                               *ListFct, ListFctMode, Bdd))
    return (BDD_MEMORY_FULL);

BListQuickDelete (&ListFctMode);

return (BDD_OK);
}

/*-----
/* GnlGetSupportOfFsmBddRec
*/
-----*/
BLIST G_BddFsmSupport;
GNL_STATUS GnlGetSupportOfFsmBddRec (Bdd)
    BDD          Bdd;
{
    BDD_PTR        BddPtr;
    int           Index;

    if (ConstantBdd (Bdd))
        return (GNL_OK);

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);
    Index = GetBddPtrIndex (BddPtr);
    if (!BListMemberOfList (G_BddFsmSupport, Index, IntIdentical))
    {
        if (BListAddElt (G_BddFsmSupport, Index))
            return (GNL_MEMORY_FULL);
    }

    if (GnlGetSupportOfFsmBddRec (GetBddPtrEdge1 (BddPtr)))
        return (GNL_MEMORY_FULL);
    if (GnlGetSupportOfFsmBddRec (GetBddPtrEdge0 (BddPtr)))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlGetSupportOfFsmBdd
*/
-----*/
/* Computes the support list of GNL_VAR belonging to the bdd 'Bdd'. */
/*-----*/
GNL_STATUS GnlGetSupportOfFsmBdd (Bdd, Support)
    BDD          Bdd;
    BLIST      *Support;
```

gnlfsmread.c

```
{

    if (BListCreateWithSize (3, &G_BddFsmSupport))
        return (GNL_MEMORY_FULL);

    if (GnlGetSupportOfFsmBddRec (Bdd))
        return (GNL_MEMORY_FULL);

    *Support = G_BddFsmSupport;
    return (GNL_OK);
}

/*
 *-----*
 * GnlGetClockCondition
 *-----*
 /* This procedure verifies that the 'clock expression' 'BddClockExpr' is
 /* of the form:
 /*      (clock'event.[clk]!clk]).<input_expr>
 /*      or      (0)
 /*      *
 /*-----*/
GNL_STATUS GnlGetClockCondition (Bdd, BddStateVar,
                                  ClockEventVar, BddClockEventId,
                                  ClockVar, BddClockId, ClockPol, Error)

    BDD          Bdd;
    BDD          BddStateVar;
    GNL_VAR     *ClockEventVar;
    int         *BddClockEventId;
    GNL_VAR     *ClockVar;
    int         *BddClockId;
    int         *ClockPol;
    int         *Error;

{
    BLIST      Support;
    int         SizeEvent;
    int         BddIdI;
    GNL_VAR    Vari;
    int         L;
    BDD          BddClkNotEvent;
    BDD          BddClkEvent;
    BDD          BddClkEventClk1;
    BDD          BddClkEventClk0;
    int         i;

    *Error = 0;

    if (GnlGetSupportOfFsmBdd (Bdd, &Support))
        return (GNL_MEMORY_FULL);

    SizeEvent = strlen ("'event");

    *ClockEventVar = NULL;
    for (i=0; i<BListSize (Support); i++)
    {
        BddIdI = (int)BListElt (Support, i);
```

gnl fsmread.c

```
VarI = GnlGetGnlVarFromBddIndex (BddIdI);

if (strlen (GnlVarName (VarI)) <= SizeEvent)
    continue;

L = strlen (GnlVarName (VarI));

/* not elegant but it works ! */  

if ((GnlVarName (VarI) [L-1] == 't') &&
    (GnlVarName (VarI) [L-2] == 'n') &&
    (GnlVarName (VarI) [L-3] == 'e') &&
    (GnlVarName (VarI) [L-4] == 'v') &&
    (GnlVarName (VarI) [L-5] == 'e') &&
    (GnlVarName (VarI) [L-6] == '\')))
{
    *BddClockEventId = BddIdI;
    *ClockEventVar = VarI;
    BListDelInsert (Support, i+1);
    break;
}
}

/* Not clock'event found. */  

if (!(*ClockEventVar))
{
    BListQuickDelete (&Support);
    *Error = 1;
    return (GNL_OK);
}

/* temporary side effect on 'GnlVarName (*ClockEventVar)'.
GnlVarName (*ClockEventVar) [L-6] = '\0';

*ClockVar = NULL;
for (i=0; i<BListSize (Support); i++)
{
    BddIdI = (int)BListElt (Support, i);
    VarI = GnlGetGnlVarFromBddIndex (BddIdI);

    if (!strcmp (GnlVarName (VarI), GnlVarName (*ClockEventVar)))
    {
        *BddClockId = BddIdI;
        *ClockVar = VarI;
        BListDelInsert (Support, i+1);
        break;
    }
}

/* Not clock found. */  

if (!(*ClockVar))
{
    BListQuickDelete (&Support);
    GnlVarName (*ClockEventVar) [L-6] = '\0';
    *Error = 1;
    return (GNL_OK);
}
```

gnlfsmread.c

```
/* Restoring the 'GnlVarName (*ClockEventVar)' original name. */
GnlVarName (*ClockEventVar) [L-6] = '\'';

BListQuickDelete (&Support);

if (bdd_cofac0 (Bdd, *BddClockEventId, &BddClkNotEvent))
    return (GNL_MEMORY_FULL);

/* It is the statevar expression if no clock'event */ 
if (BddClkNotEvent != BddStateVar)
{
    *Error = 1;
    return (GNL_OK);
}

if (bdd_cofac1 (Bdd, *BddClockEventId, &BddClkEvent))
    return (GNL_MEMORY_FULL);

if (bdd_cofac1 (BddClkEvent, *BddClockId, &BddEventClk1))
    return (GNL_MEMORY_FULL);
if (bdd_cofac0 (BddClkEvent, *BddClockId, &BddEventClk0))
    return (GNL_MEMORY_FULL);

/* At least one of both must be diffrent of 'BddStateVar' */
if ((BddEventClk1 != BddStateVar) &&
    (BddEventClk0 != BddStateVar))
{
    *Error = 1;
    return (GNL_OK);
}

if (BddEventClk1 != BddStateVar)
{
    *ClockPol = 1;
    return (GNL_OK);
}

*ClockPol = 0;

return (GNL_OK);
}

/*-----*/
/* GnlExtractClockFromBddClockCondition */ 
/*-----*/
GNL_STATUS GnlExtractClockFromBddClockCondition (BddClockCond,
                                                ClockVar, ClockPol, Error)
BDD          BddClockCond;
GNL_VAR     *ClockVar;
int         *ClockPol;
int         *Error;
{
    BLIST     Support;
    int        SizeEvent;
    GNL_VAR   ClockEventVar;
    int        i;
    int        BddIdI;
```

gnlfsmread.c

```
GNL_VAR VarI;
int L;
BDD Bdd1;
BDD Bdd2;
BDD Bdd3;
int BddClockEventId;
int BddClockId;

if (GnlGetSupportOfFsmBdd (BddClockCond, &Support))
    return (GNL_MEMORY_FULL);

/* The expression must have two variables which are the clk'event and*/
/* the clock. If not we return an error. */
if (BListSize (Support) != 2)
{
    BListQuickDelete (&Support);
    *Error = 1;
    return (GNL_OK);
}

SizeEvent = strlen ("'event");

ClockEventVar = NULL;
for (i=0; i<BListSize (Support); i++)
{
    BddIdI = (int)BListElt (Support, i);
    VarI = GnlGetGnlVarFromBddIndex (BddIdI);

    if (strlen (GnlVarName (VarI)) <= SizeEvent)
        continue;

    L = strlen (GnlVarName (VarI));

    /* not elegant but it works !
     * if ((GnlVarName (VarI) [L-1] == 't') &&
     *     (GnlVarName (VarI) [L-2] == 'n') &&
     *     (GnlVarName (VarI) [L-3] == 'e') &&
     *     (GnlVarName (VarI) [L-4] == 'v') &&
     *     (GnlVarName (VarI) [L-5] == 'e') &&
     *     (GnlVarName (VarI) [L-6] == '\''))
    {
        BddClockEventId = BddIdI;
        ClockEventVar = VarI;
        BListDelInsert (Support, i+1);
        break;
    }
}

/* Not clock'event found.
if (!ClockEventVar)
{
    BListQuickDelete (&Support);
    *Error = 1;
    return (GNL_OK);
}
```

gnlfsmread.c

```
BddClockId = (int)BListElt (Support, 0);
*ClockVar = GnlGetGnlVarFromBddIndex (BddClockId);

/* temporary side effect on 'GnlVarName (ClockEventVar)'.
GnlVarName (ClockEventVar) [L-6] = '\0';

/* Clock event and clock do not have the same name */
if (strcmp (GnlVarName (ClockEventVar), GnlVarName (*ClockVar)))
{
    GnlVarName (ClockEventVar) [L-6] = '\\';
    *Error = 1;
    return (GNL_OK);
}

GnlVarName (ClockEventVar) [L-6] = '\\';

/* We try to extract the expression :
/*           !(clk'event. [clk|!clk]) */
if (bdd_cofac0 (BddClockCond, BddClockEventId, &Bdd1))
    return (GNL_MEMORY_FULL);

if (Bdd1 != bdd_one ())
{
    BListQuickDelete (&Support);
    *Error = 1;
    return (GNL_OK);
}

/* We must have the expression :
/*           !([clk|!clk]) */
if (bdd_cofac1 (BddClockCond, BddClockEventId, &Bdd1))
    return (GNL_MEMORY_FULL);

if (bdd_cofac0 (Bdd1, BddClockId, &Bdd2))
    return (GNL_MEMORY_FULL);
if (bdd_cofac1 (Bdd1, BddClockId, &Bdd3))
    return (GNL_MEMORY_FULL);

if ((Bdd2 == bdd_one ()) && (Bdd3 == bdd_zero ()))
{
    *ClockPol = 1;
    *Error = 0;
    return (GNL_OK);
}

if ((Bdd2 == bdd_zero ()) && (Bdd3 == bdd_one ()))
{
    *ClockPol = 0;
    *Error = 0;
    return (GNL_OK);
}

*Error = 1;
return (GNL_OK);
}
```

gnlfsmread.c

```
/*-----*/
/* GnlBuildNodeFromBdd */
/*-----*/
GNL_STATUS GnlCreateInputSeqCompoVar ();
GNL_STATUS GnlBuildNodeFromBdd (Gnl, BddInput, StateVar, OutVar,
                               InputNode)
{
    GNL          Gnl;
    BDD          BddInput;
    GNL_VAR     StateVar;
    GNL_VAR     OutVar;
    GNL_NODE   *InputNode;
{
    int          Index;
    BDD_PTR    BddPtr;
    GNL_VAR    Var;
    BDD          Bdd1;
    BDD          Bdd0;
    GNL_VAR    Bdd1Var;
    GNL_VAR    Bdd0Var;
    GNL_NODE  TempNode1;
    GNL_NODE  TempNode2;
    GNL_NODE  TempNode3;
    GNL_NODE  TempNode4;
    GNL_NODE  TempNode5;
    GNL_NODE  TempNode6;
    GNL_NODE  TempNode7;
    BLIST      NewList;

    if (BddInput == bdd_zero ())
    {
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, InputNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*InputNode, (BLIST) 0);

        return (GNL_OK);
    }

    if (BddInput == bdd_one ())
    {
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, InputNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*InputNode, (BLIST) 1);

        return (GNL_OK);
    }

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (BddInput);
    Index = GetBddPtrIndex ((BDD_PTR)GetBddPtrFromBdd (BddPtr));
    Var = GnlGetGnlVarFromBddIndex (Index);

    /* We substitute the State var by the outvar */
    if (Var == StateVar)
        Var = OutVar;
    */

}
```

gnl fsmread.c

```
Bdd1 = GetBddPtrEdge1 (BddPtr);
Bdd0 = GetBddPtrEdge0 (BddPtr);

/* !var */
if ((Bdd0 == bdd_one ()) && (Bdd1 == bdd_zero ()))
{
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode1, (BLIST)Var);
    if (GnlCreateNodeNot (Gnl, TempNode1, InputNode))
        return (GNL_MEMORY_FULL);

}

/* [var (0) x] --> var.x */ 
else if (Bdd0 == bdd_zero ())
{
    if (GnlCreateInputSeqCompoVar (Gnl, Bdd1, "$b",
                                    StateVar, OutVar, &Bdd1Var))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode6))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode6, (BLIST)Bdd1Var);

    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode1, (BLIST)Var);

    if (GnlCreateNode (Gnl, GNL_AND, InputNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*InputNode, NewList);
    if (BListAddElt (NewList, (int)TempNode1))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (NewList, (int)TempNode6))
        return (GNL_MEMORY_FULL);

}
/* [var x (1)] --> x+var */ 
else if (Bdd1 == bdd_one ())
{
    if (GnlCreateInputSeqCompoVar (Gnl, Bdd0, "$b",
                                    StateVar, OutVar, &Bdd0Var))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode6))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode6, (BLIST)Bdd0Var);

    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode1, (BLIST)Var);

    if (GnlCreateNode (Gnl, GNL_OR, InputNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &NewList))
        return (GNL_MEMORY_FULL);
```

```

SetGnlNodeSons (*InputNode, NewList);
if (BListAddElt (NewList, (int)TempNode1))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode6))
    return (GNL_MEMORY_FULL);

}
/* [var x (0)] --> !var.x */  

else if (Bdd1 == bdd_zero ())
{
    if (GnlCreateInputSeqCompoVar (Gnl, Bdd0, "$b",
                                    StateVar, OutVar, &Bdd0Var))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode6))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode6, (BLIST)Bdd0Var);

    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode1, (BLIST)Var);
    if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode2))
        return (GNL_MEMORY_FULL);

    if (GnlCreateNode (Gnl, GNL_AND, InputNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*InputNode, NewList);
    if (BListAddElt (NewList, (int)TempNode2))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (NewList, (int)TempNode6))
        return (GNL_MEMORY_FULL);

}
/* [var (1) x] --> !var+x */  

else if (Bdd0 == bdd_one ())
{
    if (GnlCreateInputSeqCompoVar (Gnl, Bdd1, "$b",
                                    StateVar, OutVar, &Bdd1Var))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode6))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode6, (BLIST)Bdd1Var);

    if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (TempNode1, (BLIST)Var);
    if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode2))
        return (GNL_MEMORY_FULL);

    if (GnlCreateNode (Gnl, GNL_OR, InputNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*InputNode, NewList);
    if (BListAddElt (NewList, (int)TempNode2))
        return (GNL_MEMORY_FULL);

```

gnlfsmread.c

```
if (BListAddElt (NewList, (int)TempNode6))
    return (GNL_MEMORY_FULL);

}

else
{
/* [var x y] --> !var.x+var.y */
if (GnlCreateInputSeqCompoVar (Gnl, Bdd0, "$b",
                                StateVar, OutVar, &Bdd0Var)) /* x */
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode6))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode6, (BLIST)Bdd0Var);

if (GnlCreateInputSeqCompoVar (Gnl, Bdd1, "$b",
                                StateVar, OutVar, &Bdd1Var))/* y */
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode7))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode7, (BLIST)Bdd1Var);

if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode1))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode1, (BLIST)Var);
if (GnlCreateNodeNot (Gnl, TempNode1, &TempNode2)) /* !var */
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (Gnl, GNL_VARIABLE, &TempNode3))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode3, (BLIST)Var); /* var */ */

if (GnlCreateNode (Gnl, GNL_AND, &TempNode4)) /* !var.x */
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode4, NewList);
if (BListAddElt (NewList, (int)TempNode2))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode6))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (Gnl, GNL_AND, &TempNode5)) /* var.y */
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (TempNode5, NewList);
if (BListAddElt (NewList, (int)TempNode3))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)TempNode7))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (Gnl, GNL_OR, InputNode)) /* var.y */
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*InputNode, NewList);
if (BListAddElt (NewList, (int)TempNode4))
```

gnlfsmread.c

```
    return (GNL_MEMORY_FULL);
    if (BListAddElt (NewList, (int)TempNode5))
        return (GNL_MEMORY_FULL);

}

if (BddInput != (BDD)BddPtr)
{
    if (GnlCreateNodeNot (Gnl, *InputNode, &TempNode1))
        return (GNL_MEMORY_FULL);
    *InputNode = TempNode1;
}

return (GNL_OK);
}

/*-----
/* GnlCreateInputSeqCompoVar
/*-----
/* This procedure creates a new GNL_NODE tree corresponding to bdd
/* 'BddInput' and a new Var and new function and adds in the list of
/* functions of 'Gnl'. The new var created is returned thru '*InputVar'
/* and will constitute the input of the seq. component.
/*-----*/
GNL_STATUS GnlCreateInputSeqCompoVar (Gnl, BddInput, BaseName,
                                         StateVar, OutVar, InputVar)
{
    GNL          Gnl;
    BDD          BddInput;
    char         *BaseName;
    GNL_VAR     StateVar;
    GNL_VAR     OutVar;
    GNL_VAR     *InputVar;

    BDD_PTR     BddPtr;
    int         IndexInput;
    GNL_STATUS   GnlStatus;
    GNL_NODE    InputNode;
    GNL_VAR     NewVar;
    GNL_FUNCTION NewFunction;

    /* If the 'BddInput' corresponds to a simple signal. */
    BddPtr = (BDD_PTR)GetBddPtrFromBdd (BddInput);
    if ((GetBddPtrEdge0 (BddPtr) == bdd_zero ()) &&
        (GetBddPtrEdge1 (BddPtr) == bdd_one ()))
    {
        IndexInput = GetBddPtrIndex (BddPtr);
        *InputVar = GnlGetGnlVarFromBddIndex (IndexInput);
        if (*InputVar == StateVar)
            *InputVar = OutVar;
        return (GNL_OK);
    }

    /* If not we extract the complex node expression from it. */
    if (GnlBuildNodeFromBdd (Gnl, BddInput, StateVar, OutVar, &InputNode))
        return (GNL_MEMORY_FULL);
```

gnl fsmread.c

```
/* we create a new variable and function storing this new node */
/* expression and adds it in the current 'Gnl'. Then we return this */
/* as the actual var used in the port association with the input pin.*/
if (GnlCreateUniqueVar (Gnl, BaseName, &NewVar))
    return (GNL_MEMORY_FULL);
if (BListAddElt (GnlLocals(Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (NewVar, NewFunction);
SetGnlFunctionOnSet (NewFunction, InputNode);

if (BListAddElt (GnlFunctions(Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);

*InputVar = NewVar;

return (GNL_OK);
}

/*
 * GnlResetNodeTagRec
 */
void GnlResetNodeTagRec (Node)
    GNL_NODE Node;
{
    int             i;
    GNL_NODE SonI;
    GNL_VAR Var;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        SetGnlVarTag (Var, 0);
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        GnlResetNodeTagRec (SonI);
    }
}

/*
 * GnlVarIsEvent
 */
int GnlVarIsEvent (Var)
```

gnlfsmread.c

```
GNL_VAR Var;
{
    int L;
    char *Name;

    L = strlen (GnlVarName (Var));
    if (L <= strlen ("'event'"))
        return (0);

    Name = GnlVarName (Var)+L-strlen ("'event'");

    if (!strcmp (Name, "'event'"))
        return (1);

    return (0);
}

/*-----
 * GnlCopyGnlNode
 */
/*-----
 * This procedure make a copy of the logic cone and instantiate special */
/* signals like 'ClockVar' , 'SetVar' , 'ResetVar'. */
/*
/* pecerror =$f41
/* $f41 =mop@pecerror
/* mop@pecerror =$f166
/* $f166 =(mc@'event.$f165)
/* $f165 =((!(clk'event).$f41)+(clk'event.$f164))
/* $f164 =((!(bp0@cx(9)).$f163)+(bp0@cx(9).$f154))
/* $f163 =((!(bp0@cx(8)).$f162)+(bp0@cx(8).$f154))
/* $f162 =((!(bp0@cx(7)).$f161)+(bp0@cx(7).$f154))
/* $f161 =((!(bp0@cx(6)).$f160)+(bp0@cx(6).$f154))
/* $f160 =((!(bp0@cx(5)).$f159)+(bp0@cx(5).$f154))
/* $f159 =((!(bp0@cx(4)).$f158)+(bp0@cx(4).$f154))
/* $f158 =((!(bp0@cx(3)).$f157)+(bp0@cx(3).$f154))
/* $f157 =((!(bp0@cx(2)).$f156)+(bp0@cx(2).$f154))
/* $f156 =((!(bp0@cx(1)).$f155)+(bp0@cx(1).$f154))
/* $f155 =((!(bp0@cx(0)).!($f152))+(bp0@cx(0).$f154))
/* $f152 =((!(clk).!($f41))+(clk.$f151))
/* $f151 =(cnt2+!($f41))
/* $f154 =((!(clk).$f41)+(clk.$f153))
/* $f153 =(cnt2+$f41)
/*
/* into:
/*
/* d174 =$f41175
/* $f41175 =mop@pecerror176
/* mop@pecerror176 =$f166177
/* $f166177 =$f165178
/* $f165178 =$f164179
/* $f164179 =((!(bp0@cx(9)).$f163180)+(bp0@cx(9).$f154191))
/* $f163180 =((!(bp0@cx(8)).$f162181)+(bp0@cx(8).$f154191))
/* $f162181 =((!(bp0@cx(7)).$f161182)+(bp0@cx(7).$f154191))
/* $f161182 =((!(bp0@cx(6)).$f160183)+(bp0@cx(6).$f154191))
/* $f160183 =((!(bp0@cx(5)).$f159184)+(bp0@cx(5).$f154191))
```

gnlfsmread.c

```
/* $f159184 =((!(bp0@cx(4)).$f158185)+(bp0@cx(4).$f154191)) */  
/* $f158185 =((!(bp0@cx(3)).$f157186)+(bp0@cx(3).$f154191)) */  
/* $f157186 =((!(bp0@cx(2)).$f156187)+(bp0@cx(2).$f154191)) */  
/* $f156187 =((!(bp0@cx(1)).$f155188)+(bp0@cx(1).$f154191)) */  
/* $f155188 =((!(bp0@cx(0)).!($f152189))+(bp0@cx(0).$f154191)) */  
/* $f152189 =$f151190 */  
/* $f151190 =(cnt2+!($f41175)) */  
/* $f154191 =$f153192 */  
/* $f153192 =(cnt2+$f41175) */  
/* */  
/*-----*/  
GNL_STATUS GnlCopyGnlNode (Gnl, Node, OutVar, ClockVar, ClockPol,  
                           SetVar, AsyncSetPol, ResetVar, AsyncResetPol,  
                           *NewNode)  
{  
    GNL          Gnl;  
    GNL_NODE    Node;  
    GNL_VAR     OutVar;  
    GNL_VAR     ClockVar;  
    int         ClockPol;  
    GNL_VAR     SetVar;  
    int         AsyncSetPol;  
    GNL_VAR     ResetVar;  
    int         AsyncResetPol;  
    GNL_NODE   *NewNode;  
  
    int          i;  
    BLIST        TempList;  
    GNL_VAR     Var;  
    GNL_NODE   SonI;  
    GNL_NODE   NewSonI;  
    int          Value;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
    {  
        if (GnlNodeSons (Node))  
        {  
            if (GnlCreateNodeVdd (Gnl, NewNode))  
                return (GNL_MEMORY_FULL);  
        }  
        else  
        {  
            if (GnlCreateNodeVss (Gnl, NewNode))  
                return (GNL_MEMORY_FULL);  
        }  
        return (GNL_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR) GnlNodeSons (Node);  
  
        if (Var == ClockVar)  
        {  
            *NewNode = (GNL_NODE) ClockPol;
```

gnlfsmread.c

```
        return (GNL_OK);
    }

    if (Var == OutVar)
    {
        *NewNode = Node;
        return (GNL_OK);
    }

    if (Var == SetVar)
    {
        *NewNode = (GNL_NODE) (!AsyncSetPol);
        return (GNL_OK);
    }

    if (Var == ResetVar)
    {
        *NewNode = (GNL_NODE) (!AsyncResetPol);
        return (GNL_OK);
    }

    if (GnlVarIsEvent (Var))
    {
        *NewNode = (GNL_NODE) 1;
        return (GNL_OK);
    }

    if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))
        return (GNL_MEMORY_FULL);

    if (!GnlVarTag (Var))
        SetGnlNodeSons (*NewNode, (BLIST)Var);
    else
        SetGnlNodeSons (*NewNode, (BLIST)GnlVarTag (Var));

    return (GNL_OK);
}

if (BListCreateWithSize (BListSize (GnlNodeSons (Node)), &TempList))
    return (GNL_MEMORY_FULL);

Value = -1;
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlCopyGnlNode (Gnl, SonI, OutVar, ClockVar, ClockPol,
                        SetVar, AsyncSetPol, ResetVar, AsyncResetPol,
                        &NewSonI))
        return (GNL_MEMORY_FULL);

    if (NewSonI == (GNL_NODE) 0)
    {
        if (GnlNodeOp (Node) == GNL_AND)
        {
            Value = 0;
            break;
        }
    }
}
```

gnl fsmread.c

```
else if (GnlNodeOp (Node) == GNL_NOT)
{
    Value = 1;
    break;
}
else if (NewSonI == (GNL_NODE) 1)
{
    if (GnlNodeOp (Node) == GNL_OR)
    {
        Value = 1;
        break;
    }
    else if (GnlNodeOp (Node) == GNL_NOT)
    {
        Value = 0;
        break;
    }
}
else
{
    if (BListAddEl (TempList, (int)NewSonI))
        return (GNL_MEMORY_FULL);
}
}

if (Value != -1)
{
    BListQuickDelete (&TempList);
    *NewNode = (GNL_NODE)Value;
    return (GNL_OK);
}

if ((BListSize (TempList) == 1) && (GnlNodeOp (Node) != GNL_NOT))
{
    *NewNode = (GNL_NODE)BListEl (TempList, 0);
    BListQuickDelete (&TempList);
    return (GNL_OK);
}

if (GnlCreateNode (Gnl, GnlNodeOp (Node), NewNode))
    return (GNL_MEMORY_FULL);

SetGnlNodeSons (*NewNode, TempList);

return (GNL_OK);
}

/*-----
/* GnlPrintListFct
/*-----*/
void GnlPrintListFct (ListFct)
    BLIST      ListFct;
{
    int          i;
    GNL_VAR    VarI;
    GNL_FUNCTION FunctionI;
```

gnl fsmread.c

```
GNL_NODE NodeI;

for (i=0; i<BListSize (ListFct); i++)
{
    VarI = (GNL_VAR)BListElt (ListFct, i);
    FunctionI = GnlVarFunction (VarI);
    NodeI = GnlFunctionOnSet (FunctionI);
    fprintf (stderr, "%s =", GnlVarName (VarI));
    GnlPrintNodeRec (stderr, NodeI, 0);
    fprintf (stderr, "\n");
}
}

/*-----
/* GnlCreateNewLovalInListFct
/*-----*/
GNL_STATUS GnlCreateNewLovalInListFct (Gnl, ListFct, Var, NextNode,
                                         NewVar)
{
    GNL          Gnl;
    BLIST        ListFct;
    GNL_VAR      Var;
    GNL_NODE     NextNode;
    GNL_VAR     *NewVar;
{
    int           i;
    GNL_VAR     VarI;
    GNL_NODE     NodeI;
    GNL_FUNCTION NewFunction;

    for (i=0; i<BListSize (ListFct); i++)
    {
        VarI = (GNL_VAR)BListElt (ListFct, i);
        NodeI = GnlFunctionOnSet (GnlVarFunction (VarI));

        if (GnlEqualNode (NodeI, NextNode))
        {
            *NewVar = VarI;
            return (GNL_OK);
        }
    }

    if (GnlCreateUniqueVar (Gnl, GnlVarName (Var), NewVar))
        return (GNL_MEMORY_FULL);

    if (GnlFunctionCreate (Gnl, *NewVar, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (*NewVar, NewFunction);

    SetGnlFunctionOnSet (NewFunction, NextNode);

    if (BListAddElt (ListFct, (int) (*NewVar)))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

gnl fsmread.c

```
}

/*-----
/* GnlCopySubHierarchy
/*-----*/
/* We copy physically the subtree starting from 'Node' to break the loop*/
/* and replace the state var leaves by the corresponding output variable*/
/* The new tree is returned thru 'NewNode'.
/*-----*/
GNL_STATUS GnlCopySubHierarchy (Gnl, NewList, OutVar, StateVar,
                                Node, NewNode)
{
    GNL          Gnl;
    BLIST        NewList;
    GNL_VAR      OutVar;
    GNL_VAR      StateVar;
    GNL_NODE    Node;
    GNL_NODE *NewNode;
{
    BLIST        NewList;
    GNL_VAR      Var;
    GNL_VAR      NewVar;
    GNL_FUNCTION Function;
    GNL_NODE    NextNode;
    GNL_NODE NewNodeI;
    int         i;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        *NewNode = Node;
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (Node);

        if (GnlVarTag (Var) == (int) StateVar)
        {
            if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))
                return (GNL_MEMORY_FULL);
            SetGnlNodeSons (*NewNode, (BLIST) OutVar);
            return (GNL_OK);
        }

        if ((GnlVarTag (Var) == 0) || !GnlVarFunction (Var))
        {
            *NewNode = Node;
            return (GNL_OK);
        }

        Function = GnlVarFunction (Var);
        NextNode = GnlFunctionOnSet (Function);

        if (GnlCopySubHierarchy (Gnl, NewList, OutVar, StateVar,
                                Node, NewNode));
    }
}
```

```

        NextNode, NewNode))
    return (GNL_MEMORY_FULL);

    if (GnlCreateNewLovalInListFct (Gnl, newList, Var,
                                    *NewNode, &NewVar))
        return (GNL_MEMORY_FULL);

    if (GnlCreateNode (Gnl, GNL_VARIABLE, NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (*NewNode, (BLIST)NewVar);

    return (GNL_OK);
}

if (BListCreateWithSize (BListSize (GnlNodeSons (Node)),
                        &newList))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlCopySubHierarchy (Gnl, newList, OutVar,
                            StateVar, SonI, &NewNodeI))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (newList, (int)NewNodeI))
        return (GNL_MEMORY_FULL);
}

if (GnlCreateNode (Gnl, GnlNodeOp (Node), NewNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*NewNode, newList);

return (GNL_OK);
}

/*
/* GnlCutStateVarLoop
*/
/*
/* Loops can occur because of re-using state var (cf example below */
/* where '$f41175' is reused in '$f151190' and '$f153192' */
/*
/* d174 =$f41175
/*
/* $f41175 =mop@pecerror176
/*
/* mop@pecerror176 =$f166177
/*
/* $f166177 =$f165178
/*
/* $f165178 =$f164179
/*
/* $f164179 =((!(bp0@cx(9)).$f163180)+(bp0@cx(9).$f154191)) */
/*
/* $f163180 =((!(bp0@cx(8)).$f162181)+(bp0@cx(8).$f154191)) */
/*
/* $f162181 =((!(bp0@cx(7)).$f161182)+(bp0@cx(7).$f154191)) */
/*
/* $f161182 =((!(bp0@cx(6)).$f160183)+(bp0@cx(6).$f154191)) */
/*
/* $f160183 =((!(bp0@cx(5)).$f159184)+(bp0@cx(5).$f154191)) */
/*
/* $f159184 =((!(bp0@cx(4)).$f158185)+(bp0@cx(4).$f154191)) */
/*
/* $f158185 =((!(bp0@cx(3)).$f157186)+(bp0@cx(3).$f154191)) */
/*
/* $f157186 =((!(bp0@cx(2)).$f156187)+(bp0@cx(2).$f154191)) */
/*
/* $f156187 =((!(bp0@cx(1)).$f155188)+(bp0@cx(1).$f154191)) */

```

gnlfsmread.c

```
/* $f155188 =((!(bp0@cx(0)).!($f152189))+(bp0@cx(0).$f154191)) */  
/* $f152189 =$f151190 */  
/* $f151190 =(cnt2+!($f41175)) */  
/* $f154191 =$f153192 */  
/* $f153192 =(cnt2+$f41175) */  
/* */  
/* into: */  
/* */  
/* d174 =$f41175 */  
/* */  
/* $f41175 =mop@pecerror176 */  
/* */  
/* mop@pecerror176 =$f166177 */  
/* */  
/* $f166177 =$f165178 */  
/* */  
/* $f165178 =$f164179 */  
/* */  
/* $f164179 =((!(bp0@cx(9)).$f163180)+(bp0@cx(9).$f154191)) */  
/* */  
/* $f163180 =((!(bp0@cx(8)).$f162181)+(bp0@cx(8).$f154191)) */  
/* */  
/* $f162181 =((!(bp0@cx(7)).$f161182)+(bp0@cx(7).$f154191)) */  
/* */  
/* $f161182 =((!(bp0@cx(6)).$f160183)+(bp0@cx(6).$f154191)) */  
/* */  
/* $f160183 =((!(bp0@cx(5)).$f159184)+(bp0@cx(5).$f154191)) */  
/* */  
/* $f159184 =((!(bp0@cx(4)).$f158185)+(bp0@cx(4).$f154191)) */  
/* */  
/* $f158185 =((!(bp0@cx(3)).$f157186)+(bp0@cx(3).$f154191)) */  
/* */  
/* $f157186 =((!(bp0@cx(2)).$f156187)+(bp0@cx(2).$f154191)) */  
/* */  
/* $f156187 =((!(bp0@cx(1)).$f155188)+(bp0@cx(1).$f154191)) */  
/* */  
/* $f155188 =((!(bp0@cx(0)).!($f152189))+(bp0@cx(0).$f154191)) */  
/* */  
/* $f152189 =$f151190 */  
/* */  
/* $f151190 =(cnt2+! (pecerror)) */  
/* */  
/* $f154191 =$f153192 */  
/* */  
/* $f153192 =(cnt2+pecerror) */  
/*-----*/  
GNL_STATUS GnlCutStateVarLoop (Gnl, ListFct, NewList, Node, OutVar,  
                                StateVar, Stack, NewNode)  
{  
    GNL          Gnl;  
    BLIST        ListFct;  
    BLIST        NewList;  
    GNL_NODE     Node;  
    GNL_VAR      OutVar;  
    GNL_VAR      StateVar;  
    BLIST        Stack;  
    GNL_NODE    *NewNode;  
    int          i;  
    GNL_VAR      Var;  
    GNL_NODE     NextNode;  
    int          NewLvl;  
    GNL_NODE     SonI;  
    GNL_NODE     NewNodeI;  
    GNL_VAR      NextVar;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
    {  
        *NewNode = Node;  
        return (GNL_OK);  
    }  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR) GnlNodeSons (Node);  
    }  
}
```

```

/* This is the first point of the loop. We then copy physically */
/* the subtree to break the loop and replace the state var leaves*/
/* by the corresponding output variable */ 
if (BListMemberOfList (Stack, Var, IntIdentical))
{
    if (GnlCopySubHierarchy (Gnl, NewList, OutVar, StateVar, Node,
                           NewNode))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

if (!GnlVarFunction (Var))
{
    *NewNode = Node;
    return (GNL_OK);
}

if (!GnlVarTag (Var))
{
    *NewNode = Node;
    return (GNL_OK);
}

if (BListAddElt (Stack, (int)Var))
    return (GNL_MEMORY_FULL);

NextNode = GnlFunctionOnSet (GnlVarFunction (Var));

if (GnlCutStateVarLoop (Gnl, ListFct, NewList, NextNode, OutVar,
                       StateVar, Stack, NewNode))
    return (GNL_MEMORY_FULL);
SetGnlFunctionOnSet (GnlVarFunction (Var), *NewNode);

BSize (Stack) = BSize (Stack)-1;

*NewNode = Node;

return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlCutStateVarLoop (Gnl, ListFct, NewList, SonI, OutVar,
                           StateVar, Stack, &NewNodeI))
        return (GNL_MEMORY_FULL);
    BListElt (GnlNodeSons (Node), i) = (int)NewNodeI;
}

*NewNode = Node;

return (GNL_OK);
}

```

```

/*
 *-----*/
/* GnlBuildListFctInput */ 
/*-----*/
/* This procedure extracts the cone of logic implementing the input of */
/* the sequential component. We duplicate this cone of logic and */
/* instantiate signals like 'ClockVar', 'SetVar', ... by the value which*/
/* is the clock value and the not set/reset values. */
/* In this cone of logic loops can occur when re-using State vars so we */
/* need to break the loops. 'InputVar' will be the input attached to the*/
/* input port of the sequential component. */
/*-----*/
GNL_STATUS GnlBuildListFctInput (Gnl, OutVar, StateVar,
                                 ClockVar, ClockPol,
                                 SetVar, AsyncSetPol,
                                 ResetVar, AsyncResetPol,
                                 ListFct, InputVar)
{
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_VAR     StateVar;
    GNL_VAR     ClockVar;
    int         ClockPol;
    GNL_VAR     SetVar;
    int         AsyncSetPol;
    GNL_VAR     ResetVar;
    int         AsyncResetPol;
    BLIST      ListFct;
    GNL_VAR    *InputVar;
{
    int          i;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE    NodeI;
    GNL_VAR     NewVarI;
    GNL_FUNCTION NewFunctionI;
    GNL_NODE    NewNode;
    BLIST       Stack;
    BLIST       NewList;

#endif TRACE_EXTRACT_SEQ
fprintf (stderr, "*****\n");
GnlPrintListFct (ListFct);
#endif

for (i=0; i<BListSize (ListFct); i++)
{
    VarI = (GNL_VAR)BListElt (ListFct, i);
    FunctionI = GnlVarFunction (VarI);
    NodeI = GnlFunctionOnSet (FunctionI);

    GnlResetNodeTagRec (NodeI);
}

for (i=0; i<BListSize (ListFct); i++)
{
    VarI = (GNL_VAR)BListElt (ListFct, i);
}

```

gnlfsmread.c

```
if (VarI == OutVar)
{
    if (GnlCreateUniqueVar (Gnl, "d", &NewVarI))
        return (GNL_MEMORY_FULL);
}
else
{
    if (GnlCreateUniqueVar (Gnl, GnlVarName (VarI), &NewVarI))
        return (GNL_MEMORY_FULL);
}

SetGnlVarTag (VarI, (int)NewVarI);

SetGnlVarTag (NewVarI, (int)VarI);
}

for (i=0; i<BListSize (ListFct); i++)
{
    VarI = (GNL_VAR)BListElt (ListFct, i);
    NewVarI = (GNL_VAR)GnlVarTag (VarI);

    NodeI = GnlFunctionOnSet (GnlVarFunction (VarI));

    if (GnlCopyGnlNode (Gnl, NodeI, OutVar, ClockVar, ClockPol,
                        SetVar, AsyncSetPol, ResetVar, AsyncResetPol,
                        &NewNode))
        return (GNL_MEMORY_FULL);

    if (GnlFunctionCreate (Gnl, NewVarI, NULL, &NewFunctionI))
        return (GNL_MEMORY_FULL);

    if (NewNode == (GNL_NODE) 0)
    {
        if (GnlCreateNodeVss (Gnl, &NewNode))
            return (GNL_MEMORY_FULL);
    }
    else if (NewNode == (GNL_NODE) 1)
    {
        if (GnlCreateNodeVdd (Gnl, &NewNode))
            return (GNL_MEMORY_FULL);
    }

    SetGnlVarFunction (NewVarI, NewFunctionI);
    SetGnlFunctionOnSet (NewFunctionI, NewNode);

    BListElt (ListFct, i) = (int)NewVarI;
}

#endif TRACE_EXTRACT_SEQ
fprintf (stderr, "\n");
GnlPrintListFct (ListFct);
#endif

*InputVar = (GNL_VAR)GnlVarTag (OutVar);
GnlResetVarFunction (OutVar);
```

gnlfsmread.c

```
/* We pick up the Boolean tree functions and will cut eventually      */
/* loops created by using state var 'StateVar'.                      */
NodeI = GnlFunctionOnSet (GnlVarFunction (*InputVar));
if (BListCreate (&Stack))
    return (GNL_MEMORY_FULL);
if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
if (GnlCutStateVarLoop (Gnl, ListFct, NewList, NodeI, OutVar,
                        StateVar, Stack, &NewNode))
    return (GNL_MEMORY_FULL);
if (BListAppend (ListFct, &NewList))
    return (GNL_MEMORY_FULL);
BListQuickDelete (&Stack);

SetGnlFunctionOnSet (GnlVarFunction (*InputVar), NewNode);

#ifndef TRACE_EXTRACT_SEQ
fprintf (stderr, "\n");
GnlPrintListFct (ListFct);
fprintf (stderr, "*****\n");
#endif

for (i=0; i<BListSize (ListFct); i++)
{
    VarI = (GNL_VAR)BListElt (ListFct, i);
    if (BListAddElt (GnlFunctions (Gnl), (int)VarI))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (GnlLocals (Gnl), (int)VarI))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
}

BListQuickDelete (&ListFct);

return (GNL_OK);
}

/*
 * GnlBddIsSimpleBddRec
 */
int GnlBddIsSimpleBddRec (Bdd, Lvl)
    BDD      Bdd;
    int      Lvl;
{
    BDD_PTR  BddPtr;
    BDD      Bdd1;
    BDD      Bdd0;

    if (ConstantBdd (Bdd))
        return (1);

    if (Lvl)
        return (0);

    BddPtr = (BDD_PTR)GetBddPtrFromBdd (Bdd);
```

gnl fsmread.c

```
Bdd1 = GetBddPtrEdge1 (BddPtr) ;
Bdd0 = GetBddPtrEdge0 (BddPtr) ;

if (!GnlBddIsSimpleBddRec (Bdd1, 1))
    return (0);

if (!GnlBddIsSimpleBddRec (Bdd0, 1))
    return (0);

return (1);
}

/*
/* GnlBddIsSimpleBdd
*/
int GnlBddIsSimpleBdd (Bdd)
    BDD      Bdd;
{
    return (GnlBddIsSimpleBddRec (Bdd, 0));
}

/*
/* GnlCreateFsmSeqComponent
*/
/* This procedure creates a new GNL_SEQUENTIAL_COMPONENT (and
/* GNL_COMPONENT then) and adds it in the list of Components of the
/* 'Gnl'.
*/
GNL_STATUS GnlCreateFsmSeqComponent (Gnl, TypeSeqElem, OutVar, StateVar,
                                      BddInput, ListFct,
                                      ClockVar, ClockPol,
                                      AsyncSet, AsyncSetPol,
                                      AsyncReset, AsyncResetPol,
                                      SetPriorReset)
{
    GNL           Gnl;
    int           TypeSeqElem;      /* 0 --> latch, 1 --> Dff          */
    GNL_VAR       OutVar;
    GNL_VAR       StateVar;
    BDD           BddInput;
    BLIST         ListFct;
    GNL_VAR       ClockVar;
    int           ClockPol;
    int           AsyncSet;
    int           AsyncSetPol;
    int           AsyncReset;
    int           AsyncResetPol;
    int           SetPriorReset;
{
    BLIST           Components;
    GNL_VAR         SetVar;
    GNL_VAR         ResetVar;
    GNL_SEQUENTIAL_COMPONENT NewCompo;
    char            *InstName;
    BLIST           newList;
    GNL_VAR         InputVar;
```

```

/* Creating eventually the list of components. */  

if (!GnlComponents (Gnl))  

{  

    if (BListCreate (&NewList))  

        return (GNL_MEMORY_FULL);  

    SetGnlComponents (Gnl, NewList);  

}  

Components = GnlComponents (Gnl);  

SetVar = NULL;  

if (AsyncSet > 0)  

    SetVar = GnlGetGnlVarFromBddIndex (AsyncSet);  

ResetVar = NULL;  

if (AsyncReset > 0)  

    ResetVar = GnlGetGnlVarFromBddIndex (AsyncReset);  

/* If the bdd input expression is a simple Bdd (either a constante */  

/* or a simple variable) then we generates equations from this Bdd */  

/* otherwise we generate equations by instantiating the original cone*/  

/* of logic. */  

if (GnlBddIsSimpleBdd (BddInput))  

{  

    if (GnlCreateInputSeqCompoVar (Gnl, BddInput, "d", StateVar,  

                                    OutVar, &InputVar))  

        return (GNL_MEMORY_FULL);  

}  

else  

{  

    if (GnlBuildListFctInput (Gnl, OutVar, StateVar, ClockVar, ClockPol,  

                            SetVar, AsyncSetPol, ResetVar, AsyncResetPol,  

                            ListFct, &InputVar))  

        return (GNL_MEMORY_FULL);  

}  

if (SetVar)  

{  

    /* if the set var direction can be erased. */  

    if (!GnlVarIsPrimary (SetVar))  

        SetGnlVarDir (SetVar, GNL_VAR_LOCAL_WIRING);  

}  

if (ResetVar)  

{  

    /* if the reset var direction can be erased. */  

    if (!GnlVarIsPrimary (ResetVar))  

        SetGnlVarDir (ResetVar, GNL_VAR_LOCAL_WIRING);  

}  

if (ClockVar)  

{  

    /* if the clock var direction can be erased. */  

    if (!GnlVarIsPrimary (ClockVar))  

        SetGnlVarDir (ClockVar, GNL_VAR_LOCAL_WIRING);  

}

```

```

if (InputVar)
{
    /* if the input var direction can be erased. */
    if (!GnlVarIsPrimary (InputVar))
        SetGnlVarDir (InputVar, GNL_VAR_LOCAL_WIRING);
}

if (OutVar)
{
    /* if the output var direction can be erased. */
    if (!GnlVarIsPrimary (OutVar))
        SetGnlVarDir (OutVar, GNL_VAR_LOCAL_WIRING);
}

/* Flip-Flop with Async. Set/Reset. */
if (SetVar && ResetVar)
{
/*
    if (TypeSeqElem)
        fprintf (stderr, " Generate a DFF Reset/Set/%d for <%s>\n",
                SetPriorReset, GnlVarName (OutVar));
    else
        fprintf (stderr, " Generate a LATCH Reset/Set/%d for <%s>\n",
                SetPriorReset, GnlVarName (OutVar));
*/
    if (GnlCreateSequentialComponent (GNL_DFF, &NewCompo))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (Components, (int)NewCompo))
        return (GNL_MEMORY_FULL);
    if (GnlStrCopy (GnlVarName (OutVar), &InstName))
        return (GNL_MEMORY_FULL);
    SetGnlSequentialCompoInstName (NewCompo, InstName);

    SetGnlSequentialCompoInput (NewCompo, InputVar);
    SetGnlSequentialCompoOutput (NewCompo, OutVar);
    SetGnlSequentialCompoClock (NewCompo, ClockVar);
    SetGnlSequentialCompoClockPol (NewCompo, ClockPol);
    SetGnlSequentialCompoReset (NewCompo, ResetVar);
    SetGnlSequentialCompoResetPol (NewCompo, AsyncResetPol);
    SetGnlSequentialCompoSet (NewCompo, SetVar);
    SetGnlSequentialCompoSetPol (NewCompo, AsyncSetPol);

    switch (SetPriorReset) {
        case 0:
            if (TypeSeqElem)
                SetGnlSequentialCompoOp (NewCompo, GNL_DFF0);
            else
                SetGnlSequentialCompoOp (NewCompo, GNL_LATCH0);
            break;
        case 1:
            if (TypeSeqElem)
                SetGnlSequentialCompoOp (NewCompo, GNL_DFF1);
            else
                SetGnlSequentialCompoOp (NewCompo, GNL_LATCH1);
            break;
        default:
            if (TypeSeqElem)

```

```

        SetGnlSequentialCompoOp (NewCompo, GNL_DFFX) ;
    else
        SetGnlSequentialCompoOp (NewCompo, GNL_LATCHX) ;
    }

    return (GNL_OK) ;
}

/* Flip-Flop with Async. Set
if (SetVar)
{
/*
    if (TypeSeqElem)
        fprintf (stderr, " Generate a DFF Set for <%s>\n",
                 GnlVarName (OutVar));
    else
        fprintf (stderr, " Generate a LATCH Set for <%s>\n",
                 GnlVarName (OutVar));
*/
    if (GnlCreateSequentialComponent (GNL_DFF, &NewCompo))
        return (GNL_MEMORY_FULL);
    if (!TypeSeqElem)
        SetGnlSequentialCompoOp (NewCompo, GNL_LATCH);
    if (BListAddElt (Components, (int)NewCompo))
        return (GNL_MEMORY_FULL);
    if (GnlStrCopy (GnlVarName (OutVar), &InstName))
        return (GNL_MEMORY_FULL);
    SetGnlSequentialCompoInstName (NewCompo, InstName);

    SetGnlSequentialCompoInput (NewCompo, InputVar);
    SetGnlSequentialCompoOutput (NewCompo, OutVar);
    SetGnlSequentialCompoClock (NewCompo, ClockVar);
    SetGnlSequentialCompoClockPol (NewCompo, ClockPol);
    SetGnlSequentialCompoSet (NewCompo, SetVar);
    SetGnlSequentialCompoSetPol (NewCompo, AsyncSetPol);

    return (GNL_OK);
}

/* Flip-Flop with Async. Reset
if (ResetVar)
{
/*
    if (TypeSeqElem)
        fprintf (stderr, " Generate a DFF Reset for <%s>\n",
                 GnlVarName (OutVar));
    else
        fprintf (stderr, " Generate a LATCH Reset for <%s>\n",
                 GnlVarName (OutVar));
*/
    if (GnlCreateSequentialComponent (GNL_DFF, &NewCompo))
        return (GNL_MEMORY_FULL);
    if (!TypeSeqElem)
        SetGnlSequentialCompoOp (NewCompo, GNL_LATCH);
    if (BListAddElt (Components, (int)NewCompo))
        return (GNL_MEMORY_FULL);
    if (GnlStrCopy (GnlVarName (OutVar), &InstName))

```

gnl fsmread.c

```
        return (GNL_MEMORY_FULL);
SetGnlSequentialCompoInstName (NewCompo, InstName);

SetGnlSequentialCompoInput (NewCompo, InputVar);
SetGnlSequentialCompoOutput (NewCompo, OutVar);
SetGnlSequentialCompoClock (NewCompo, ClockVar);
SetGnlSequentialCompoClockPol (NewCompo, ClockPol);
SetGnlSequentialCompoReset (NewCompo, ResetVar);
SetGnlSequentialCompoResetPol (NewCompo, AsyncResetPol);

        return (GNL_OK);
    }

/* Basic Flip-Flop */  

/*  

if (TypeSeqElem)
    fprintf (stderr, " Generate a DFF for <%s>\n",
            GnlVarName (OutVar));
else
    fprintf (stderr, " Generate a LATCH for <%s>\n",
            GnlVarName (OutVar));
*/
if (GnlCreateSequentialComponent (GNL_DFF, &NewCompo))
    return (GNL_MEMORY_FULL);
if (!TypeSeqElem)
    SetGnlSequentialCompoOp (NewCompo, GNL_LATCH);
if (BListAddElt (Components, (int)NewCompo))
    return (GNL_MEMORY_FULL);
if (GnlStrCopy (GnlVarName (OutVar), &InstName))
    return (GNL_MEMORY_FULL);
SetGnlSequentialCompoInstName (NewCompo, InstName);

SetGnlSequentialCompoInput (NewCompo, InputVar);
SetGnlSequentialCompoOutput (NewCompo, OutVar);
SetGnlSequentialCompoClock (NewCompo, ClockVar);
SetGnlSequentialCompoClockPol (NewCompo, ClockPol);

        return (GNL_OK);
}

/*-----*/
/* GnlClockEventVar */  

/*-----*/
int GnlClockEventVar (Var)
    GNL_VAR Var;
{
    int      L;
    int      SizeEvent;

    SizeEvent = strlen ("event");
    L = strlen (GnlVarName (Var));

    if (strlen (GnlVarName (Var)) <= SizeEvent)
        return (0);
```

gnl fsmread.c

```
/* not elegant but it works ! */  
if ((GnlVarName (Var) [L-1] == 't') &&  
    (GnlVarName (Var) [L-2] == 'n') &&  
    (GnlVarName (Var) [L-3] == 'e') &&  
    (GnlVarName (Var) [L-4] == 'v') &&  
    (GnlVarName (Var) [L-5] == 'e') &&  
    (GnlVarName (Var) [L-6] == '\')))  
{  
    return (1);  
}  
  
return (0);  
}  
  
/*-----*/  
/* GnlExtractDffSeqCompo */  
/*-----*/  
/* This procedure checks if we can infer a Dff from the bdd expression */  
/* in 'Bdd'. */  
/*-----*/  
GNL_STATUS GnlExtractDffSeqCompo (Gnl, OutVar, StateVar, Bdd, Support,  
                                  BddInput, ClockVar, ClockPol, AsyncSet,  
                                  AsyncSetPol, AsyncReset, AsyncResetPol,  
                                  SetPriorReset, Error)  
{  
    GNL      Gnl;  
    GNL_VAR  OutVar;  
    GNL_VAR  StateVar;  
    BDD      Bdd;  
    BDD      *BddInput;  
    BLIST   Support;  
    GNL_VAR *ClockVar;  
    int      *ClockPol;  
    int      *AsyncSet;  
    int      *AsyncSetPol;  
    int      *AsyncReset;  
    int      *AsyncResetPol;  
    int      *SetPriorReset;  
    int      *Error;  
  
    int      i;  
    BDD      RealBdd;  
    int      IndexBddI;  
    GNL_VAR CofVar;  
    BDD      Bdd1;  
    BDD      Bdd2;  
    BDD      Bdd3;  
    BDD      Bdd4;  
    BDD      BddStateVar;  
    int      IndexBddStateVar;  
    BDD      BddClockExpr;  
    GNL_VAR ClockEventVar;  
    int      BddClockEventId;  
    int      BddClockId;  
    BDD      BddStateVarExpr;  
  
    *Error = 0;
```

```

/* Now we analyze the 'RealBdd' expression and extract in the */
/* following steps:                                         */
/*   - extraction of asynchronous signals (Set/Reset)      */
/* cases are:                                              */
/* Cofactor (RealBdd|x = 1) = 0 ==> X Async. Reset, Polarity 1 */
/* Cofactor (RealBdd|x = 1) = 1 ==> X Async. Set,   Polarity 1 */
/* Cofactor (RealBdd|x = 0) = 0 ==> X Async. Reset, Polarity 0 */
/* Cofactor (RealBdd|x = 0) = 1 ==> X Async. Set,   Polarity 0 */

/* For now, no asynchronous reset/set extracted.           */
*AsyncSet = -1;
*AsyncReset = -1;

/* We extract the prioritary asynchronous signal which is either a */
/* Set or a Reset.                                         */
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);

    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (CofVar == StateVar)
        continue;

    if (GnlClockEventVar (CofVar))
        continue;

    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);

    /* We extracted an asynchronous Set                      */
    if (RealBdd == bdd_one ())
    {
        *AsyncSet = IndexBddI;
        *AsyncSetPol = 1;
        BListDelInsert (Support, i+1);
        /* We remove the asynchronous Set                     */
        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);
        *SetPriorReset = 1;
        Bdd = RealBdd;
        break;
    }
    else if (RealBdd == bdd_zero ())
    {
        /* We extracted an asynchronous Reset.             */
        *AsyncReset = IndexBddI;
        *AsyncResetPol = 1;
        BListDelInsert (Support, i+1);
        /* We remove the asynchronous Reset               */
        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);
        *SetPriorReset = 0;
        Bdd = RealBdd;
        break;
    }
}

```

```

if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
    return (GNL_MEMORY_FULL);

/* We extracted an asynchronous Set */
if (RealBdd == bdd_one ())
{
    *AsyncSet = IndexBddI;
    *AsyncSetPol = 0;
    BListDelInsert (Support, i+1);
    /* We remove the asynchronous Set */
    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))           */
        return (GNL_MEMORY_FULL);
    *SetPriorReset = 1;
    Bdd = RealBdd;
    break;
}
else if (RealBdd == bdd_zero ())
{
    /* We extracted an asynchronous Reset. */
    *AsyncReset = IndexBddI;
    *AsyncResetPol = 0;
    BListDelInsert (Support, i+1);
    /* We remove the asynchronous Reset */
    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))           */
        return (GNL_MEMORY_FULL);
    *SetPriorReset = 0;
    Bdd = RealBdd;
    break;
}
}

/* 'Bdd' is the current bdd expression and may have been modified. */
/* We previously extracted an 'AsyncSet' or an 'AsyncReset' and then */
/* we look for the second one (in case we have both Set and Reset). */
if ((*AsyncSet > 0) || (*AsyncReset > 0))
{
    for (i=0; i<BListSize (Support); i++)
    {
        IndexBddI = (int)BListElt (Support, i);

        CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
        if (CofVar == StateVar)
            continue;

        if (GnlClockEventVar (CofVar))
            continue;

        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);

        /* We extracted an asynchronous Set */
        if (RealBdd == bdd_one ())
        {
            *AsyncSet = IndexBddI;
            *AsyncSetPol = 1;
            BListDelInsert (Support, i+1);
        }
    }
}

```

```

        /* We remove the asynchronous Set */  

        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

            return (GNL_MEMORY_FULL);  

        Bdd = RealBdd;  

        break;  

    }  

    else if (RealBdd == bdd_zero ())  

    {  

        /* We extracted an asynchronous Reset. */  

        *AsyncReset = IndexBddI;  

        *AsyncResetPol = 1;  

        BListDelInsert (Support, i+1);  

        /* We remove the asynchronous Reset */  

        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

            return (GNL_MEMORY_FULL);  

        Bdd = RealBdd;  

        break;  

    }  

    if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

        return (GNL_MEMORY_FULL);  

    /* We extracted an asynchronous Set */  

    if (RealBdd == bdd_one ())  

    {  

        *AsyncSet = IndexBddI;  

        *AsyncSetPol = 0;  

        BListDelInsert (Support, i+1);  

        /* We remove the asynchronous Set */  

        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))  

            return (GNL_MEMORY_FULL);  

        Bdd = RealBdd;  

        break;  

    }  

    else if (RealBdd == bdd_zero ())  

    {  

        /* We extracted an asynchronous Reset. */  

        *AsyncReset = IndexBddI;  

        *AsyncResetPol = 0;  

        BListDelInsert (Support, i+1);  

        /* We remove the asynchronous Reset */  

        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))  

            return (GNL_MEMORY_FULL);  

        Bdd = RealBdd;  

        break;  

    }  

    }  

}  

/* 'Bdd' is the current bdd expression and may have been modified. */  

/* We have extracted the Asynchronous signals. Now we should analyze */  

/* a Bdd expression of the form: */  

/* StateVar <= !<clk_cond>.StateVar+<clk_cond>.<input_expr> */  

/* where <clk_cond> = (clk'event.[clk]!clk]) and <input_expr> any */  


```

gnlfsmread.c

```
/* Boolean expression. */  
/*  
BddStateVar = (BDD)GnlVarHook (StateVar);  
  
/* we extract the clock condition which is of the form: */  
/* <clk_cond> = (clk'event.[clk|!clk]) */  
if (GnlGetClockCondition (Bdd, BddStateVar,  
                           &ClockEventVar, &BddClockEventId,  
                           ClockVar, &BddClockId, ClockPol, Error))  
    return (GNL_MEMORY_FULL);  
  
/* we were not able to extract any clock */  
if (*Error)  
{  
    BListQuickDelete (&Support);  
    return (GNL_OK);  
}  
  
/* Now we get the expression: !<clk_cond>+<clk_cond>. <input_expr> */  
/* by cofactoring the Bdd expression by 'StateVar = 1' */  
if (bdd_cofac1 (Bdd, BddClockEventId, &Bdd1))  
    return (GNL_MEMORY_FULL);  
if (*ClockPol)  
{  
    if (bdd_cofac1 (Bdd1, BddClockId, BddInput))  
        return (GNL_MEMORY_FULL);  
}  
else  
{  
    if (bdd_cofac0 (Bdd1, BddClockId, BddInput))  
        return (GNL_MEMORY_FULL);  
}  
  
BListQuickDelete (&Support);  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlExtractLatchSeqCompo */  
/*-----*/  
/* This procedure checks if we can infer a Latch from the bdd expression*/  
/* in 'Bdd'. */  
/*-----*/  
GNL_STATUS GnlExtractLatchSeqCompo (Gnl, OutVar, StateVar, Bdd,  
                                      BddInput, ClockVar, ClockPol, AsyncSet,  
                                      AsyncSetPol, AsyncReset, AsyncResetPol,  
                                      SetPriorReset, Error)  
{  
    GNL          Gnl;  
    GNL_VAR     OutVar;  
    GNL_VAR     StateVar;  
    BDD         Bdd;  
    BDD         *BddInput;  
    GNL_VAR     *ClockVar;  
    int         *ClockPol;
```

gnlfsmread.c

```
int          *AsyncSet;
int          *AsyncSetPol;
int          *AsyncReset;
int          *AsyncResetPol;
int          *SetPriorReset;
int          *Error;

{
    int          i;
    BDD          RealBdd;
    int          IndexBddI;
    GNL_VAR      CofVar;
    BDD          Bdd1;
    BDD          Bdd2;
    BDD          Bdd3;
    BDD          Bdd4;
    BDD          BddStateVar;
    BDD          BddClockExpr;
    int          BddClockId;
    BDD          BddStateVarExpr;
    BLIST        Support;
    int          IndexStateVar;
    GNL_STATUS   GnlStatus;

    if (GnlGetSupportOfFsmBdd (Bdd, &Support))
        return (GNL_MEMORY_FULL);

    *Error = 0;

    /* No clock'event variable here */
    for (i=0; i<BListSize (Support); i++)
    {
        IndexBddI = (int)BListElt (Support, i);

        CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
        if (GnlClockEventVar (CofVar))
        {
            *Error = 1;
            BListQuickDelete (&Support);
            return (GNL_OK);
        }
    }

    /* Now we analyze the 'RealBdd' expression and extract in the */
    /* following steps: */
    /*           - extraction of asynchronous signals (Set/Reset) */
    /* cases are: */
    /*     Cofactor (RealBdd|x = 1) = 0 ==> X Async. Reset, Polarity 1 */
    /*     Cofactor (RealBdd|x = 1) = 1 ==> X Async. Set,    Polarity 1 */
    /*     Cofactor (RealBdd|x = 0) = 0 ==> X Async. Reset, Polarity 0 */
    /*     Cofactor (RealBdd|x = 0) = 1 ==> X Async. Set,    Polarity 0 */

    /* For now, no asynchronous reset/set extracted. */
    *AsyncSet = -1;
    *AsyncReset = -1;

    /* We extract the priotary asynchronous signal which is either a */

```

```

/* Set or a Reset.                                     */
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);

    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (CofVar == StateVar)
        continue;

    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);

    /* We extracted an asynchronous Set                 */
    if (RealBdd == bdd_one ())
    {
        *AsyncSet = IndexBddI;
        *AsyncSetPol = 1;
        BListDelInsert (Support, i+1);
        /* We remove the asynchronous Set                */
        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))   */
            return (GNL_MEMORY_FULL);
        *SetPriorReset = 1;
        Bdd = RealBdd;
        break;
    }
    else if (RealBdd == bdd_zero ())
    {
        /* We extracted an asynchronous Reset.          */
        *AsyncReset = IndexBddI;
        *AsyncResetPol = 1;
        BListDelInsert (Support, i+1);
        /* We remove the asynchronous Reset             */
        if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))   */
            return (GNL_MEMORY_FULL);
        *SetPriorReset = 0;
        Bdd = RealBdd;
        break;
    }

    if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);

    /* We extracted an asynchronous Set                 */
    if (RealBdd == bdd_one ())
    {
        *AsyncSet = IndexBddI;
        *AsyncSetPol = 0;
        BListDelInsert (Support, i+1);
        /* We remove the asynchronous Set                */
        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))   */
            return (GNL_MEMORY_FULL);
        *SetPriorReset = 1;
        Bdd = RealBdd;
        break;
    }
    else if (RealBdd == bdd_zero ())
    {

```

```

/* We extracted an asynchronous Reset. */  

*AsyncReset = IndexBddI;  

*AsyncResetPol = 0;  

BListDelInsert (Support, i+1);  

/* We remove the asynchronous Reset */  

if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))  

    return (GNL_MEMORY_FULL);  

*SetPriorReset = 0;  

Bdd = RealBdd;  

break;  

}  

}  

/* 'Bdd' is the current bdd expression and may have been modified. */  

/* We previously extracted an 'AsyncSet' or an 'AsyncReset' and then */  

/* we look for the second one (in case we have both Set and Reset). */  

if ((*AsyncSet > 0) || (*AsyncReset > 0))  

{
    for (i=0; i<BListSize (Support); i++)  

    {
        IndexBddI = (int)BListElt (Support, i);  

        CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);  

        if (CofVar == StateVar)  

            continue;  

        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))  

            return (GNL_MEMORY_FULL);  

        /* We extracted an asynchronous Set */  

        if (RealBdd == bdd_one ())  

        {
            *AsyncSet = IndexBddI;  

            *AsyncSetPol = 1;  

            BListDelInsert (Support, i+1);  

            /* We remove the asynchronous Set */  

            if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

                return (GNL_MEMORY_FULL);  

            Bdd = RealBdd;  

            break;
        }
        else if (RealBdd == bdd_zero ())  

        {
            /* We extracted an asynchronous Reset. */  

            *AsyncReset = IndexBddI;  

            *AsyncResetPol = 1;  

            BListDelInsert (Support, i+1);  

            /* We remove the asynchronous Reset */  

            if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

                return (GNL_MEMORY_FULL);  

            Bdd = RealBdd;  

            break;
        }
    }
    if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))  

        return (GNL_MEMORY_FULL);

```

```

/* We extracted an asynchronous Set */ 
if (RealBdd == bdd_one ())
{
    *AsyncSet = IndexBddI;
    *AsyncSetPol = 0;
    BListDelInsert (Support, i+1);
    /* We remove the asynchronous Set */
    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);
    Bdd = RealBdd;
    break;
}
else if (RealBdd == bdd_zero ())
{
    /* We extracted an asynchronous Reset. */ 
    *AsyncReset = IndexBddI;
    *AsyncResetPol = 0;
    BListDelInsert (Support, i+1);
    /* We remove the asynchronous Reset */
    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);
    Bdd = RealBdd;
    break;
}

/* 'Bdd' is the current bdd expression and may have been modified. */ 

/* We have extracted the Asynchronous signals. Now we should analyze */
/* a Bdd expression of the form: */ 
/* StateVar <= !<clk_cond>.StateVar+<clk_cond>.<input_expr> */ 
/* where <clk_cond> = ([clk]!clk]) and <input_expr> any */ 
/* Boolean expression. */ 
/* */

BddStateVar = (BDD)GnlVarHook (StateVar);

/* We look for the signal which by cofactoring it to 1 or 0 gives the */
/* expression of <StateVar>. */ 
*ClockVar = NULL;
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);

    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (CofVar == StateVar)
        continue;

    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);

    /* This is the clock with a low level polarity */
    if (RealBdd == BddStateVar)

```

```

{
    *ClockVar = CofVar;
    *ClockPol = 0;

    /* We remove the clock */ 
    if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);
    Bdd = RealBdd;
    break;
}

if (bdd_cofac0 (Bdd, IndexBddI, &RealBdd))
    return (GNL_MEMORY_FULL);

/* This is the clock with a High level polarity */
if (RealBdd == BddStateVar)
{
    *ClockVar = CofVar;
    *ClockPol = 1;

    /* We remove the clock */ 
    if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
        return (GNL_MEMORY_FULL);
    Bdd = RealBdd;
    break;
}
}

/* we did not find any clock */
if (*ClockVar == NULL)
{
    IndexStateVar = -1;
    for (i=0; i<BListSize (Support); i++)
    {
        IndexBddI = (int)BListElt (Support, i);
        CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
        if (CofVar == StateVar)
        {
            IndexStateVar = IndexBddI;
            break;
        }
    }

    if (IndexStateVar == -1)
    {
        *Error = 1;
        BListQuickDelete (&Support);
        return (GNL_OK);
    }
}

if (bdd_cofac1 (Bdd, IndexStateVar, &RealBdd))
    return (GNL_MEMORY_FULL);

if (RealBdd != bdd_one ())
{
    *Error = 1;
    BListQuickDelete (&Support);
}

```

```

        return (GNL_OK);
    }

    if ((GnlStatus = GnlVarCreateAndAddInHashTable (Gnl, "0",
                                                    ClockVar))) {
        if (GnlStatus == GNL_MEMORY_FULL)
            return (GNL_MEMORY_FULL);
    }
    *ClockPol = 1;
    *BddInput = RealBdd;

    BListQuickDelete (&Support);

    return (GNL_OK);
}

*BddInput = Bdd;

BListQuickDelete (&Support);

return (GNL_OK);
}

/*
 * GnlExtractSeqCompoFromBdd
 */
/*
 * This procedure analyzes the bdd 'Bdd' and extracts all the relative *
 * expression to the universal seq. element like 'clock', 'Set', ...   */
/* The bdd must be of the form:                                         */
/*     mc'event.[RealBdd]                                              */
/*
 *-----*/
GNL_STATUS GnlExtractSeqCompoFromBdd (Gnl, OutVar, StateVar, Bdd, ListFct)
    GNL          Gnl;
    GNL_VAR     OutVar;
    GNL_VAR     StateVar;
    BDD         Bdd;
    BLIST       ListFct;
{
    BLIST       Support;
    int         i;
    int         AsyncSet;
    int         AsyncSet1;
    int         AsyncSet2;
    int         AsyncReset;
    int         AsyncReset1;
    int         AsyncReset2;
    int         IndexBddI;
    GNL_VAR    CofVar;
    int         IndexBddStateVar;
    BDD        BddClockExpr;
    BDD        BddStateVar;
    GNL_VAR    ClockVar;
    int         ClockPol;
    GNL_VAR    ClockVar1;
    int         ClockPol1;
    GNL_VAR    ClockVar2;

```

```

int          ClockPol2;
BDD          BddStateVarExpr;
BDD          BddInput;
BDD          RealBdd;
GNL_VAR    ClockEventVar;
int          BddClockEventId;
int          BddClockId;
int          SetPriorReset;
int          SetPriorReset1;
int          SetPriorReset2;
int          Error;
int          AsyncSetPol;
int          AsyncResetPol;
int          AsyncSetPol1;
int          AsyncResetPol1;
int          AsyncSetPol2;
int          AsyncResetPol2;
int          NbDontcare;
GNL_VAR    DontCareVar;
int          DontCareId;
BDD          BddInput1;
BDD          BddInput2;
BDD          Bdd1;
BDD          Bdd2;
GNL_STATUS  Status;

if (GnlGetSupportOffFsmBdd (Bdd, &Support))
    return (GNL_MEMORY_FULL);

/* First we scan the support in order to remove the master clock of */
/* name GNL_MASTER_CLOCK_EVENT. We get then the Bdd 'RealBdd'. */
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);
    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (!strcmp (GnlVarName (CofVar), GNL_MASTER_CLOCK_EVENT))
    {
        if (bdd_cofac1 (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);
        BListDelInsert (Support, i+1);
        Bdd = RealBdd;
        break;
    }
}

/* We scan the support in order to extract dontcare. */ 
NbDontcare = 0;
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);
    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (GnlVarDir (CofVar) == GNL_VAR_LOCAL_DONTCARE)
    {
        /* we accept only 1 single dont care in the expression */
        if (NbDontcare)
        {

```

```

        fprintf (stderr,
                  " ERROR: state variable <%s> cannot be synthesized\n",
                  GnlVarName (OutVar));
        return (GNL_BAD_FSM_FOR_SYNTHESIS);
    }
NbDontcare++;

DontCareVar = CofVar;
DontCareId = IndexBddI;
}
}

if (NbDontcare)
{
    BListQuickDelete (&Support);

    /* we set dontcare to 1 and extract the sequential component */
    if (bdd_cofac1 (Bdd, DontCareId, &Bdd1))
        return (GNL_MEMORY_FULL);

    if (GnlGetSupportOfFsmBdd (Bdd1, &Support))
        return (GNL_MEMORY_FULL);
    if ((Status = GnlExtractDffSeqCompo (Gnl, OutVar, StateVar, Bdd1,
                                          Support, &BddInput1, &ClockVar1, &ClockPol1,
                                          &AsyncSet1, &AsyncSetPol1, &AsyncReset1,
                                          &AsyncResetPol1, &SetPriorReset1, &Error)))
        return (Status);
    BListQuickDelete (&Support);

    if (!Error)
    {
        /* we set dontcare to 0 and extract the sequential component */
        if (bdd_cofac0 (Bdd, DontCareId, &Bdd2))
            return (GNL_MEMORY_FULL);

        if (GnlGetSupportOfFsmBdd (Bdd2, &Support))
            return (GNL_MEMORY_FULL);
        if ((Status = GnlExtractDffSeqCompo (Gnl, OutVar, StateVar, Bdd2,
                                              Support, &BddInput2, &ClockVar2, &ClockPol2,
                                              &AsyncSet2, &AsyncSetPol2, &AsyncReset2,
                                              &AsyncResetPol2, &SetPriorReset2, &Error)))
            return (Status);
        BListQuickDelete (&Support);

        if (!Error &&
            (BddInput1 == BddInput2) &&
            (ClockVar1 == ClockVar2) &&
            (ClockPol1 == ClockPol2) &&
            (AsyncSet1 == AsyncSet2) &&
            (AsyncSetPol1 == AsyncSetPol2) &&
            (AsyncReset1 == AsyncReset2) &&
            (AsyncResetPol1 == AsyncResetPol2) &&
            (SetPriorReset1 != SetPriorReset2))
        {
            /* Actually there is no real priority between Set and      */
            /* Reset. The priority depends on the dontcare value.      */
            SetPriorReset = 2;
        }
    }
}

```

```

/* Now we create physically a new GNL_SEQUENTIAL_COMPONENT*/
/* in the list of components of the current 'Gnl'.          */
if (GnlCreateFsmSeqComponent (Gnl, 1, OutVar, StateVar,
                             BddInput1, ListFct,
                             ClockVar1, ClockPol1,
                             AsyncSet1, AsyncSetPol1,
                             AsyncReset1, AsyncResetPol1,
                             SetPriorReset))
    return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
}

/* If it is not good we try to extract a latch           */
/* we set dontcare to 1 and extract the sequential component      */
if (bdd_cofac1 (Bdd, DontCareId, &Bdd1))
    return (GNL_MEMORY_FULL);

if ((Status = GnlExtractLatchSeqCompo (Gnl, OutVar, StateVar, Bdd1,
                                         &BddInput1, &ClockVar1, &ClockPol1,
                                         &AsyncSet1, &AsyncSetPol1, &AsyncReset1,
                                         &AsyncResetPol1, &SetPriorReset1, &Error)))
    return (Status);
BListQuickDelete (&Support);

if (Error)
{
    fprintf (stderr,
             " ERROR: state variable <%s> cannot be synthesized\n",
             GnlVarName (OutVar));
    return (GNL_BAD_FSM_FOR_SYNTHESIS);
}

/* we set dontcare to 0 and extract the sequential component      */
if (bdd_cofac0 (Bdd, DontCareId, &Bdd2))
    return (GNL_MEMORY_FULL);

if ((Status = GnlExtractLatchSeqCompo (Gnl, OutVar, StateVar, Bdd2,
                                         &BddInput2, &ClockVar2, &ClockPol2,
                                         &AsyncSet2, &AsyncSetPol2, &AsyncReset2,
                                         &AsyncResetPol2, &SetPriorReset2, &Error)))
    return (Status);
BListQuickDelete (&Support);

if (Error ||
    (BddInput1 != BddInput2) ||
    (ClockVar1 != ClockVar2) ||
    (ClockPol1 != ClockPol2) ||
    (AsyncSet1 != AsyncSet2) ||
    (AsyncSetPol1 != AsyncSetPol2) ||
    (AsyncReset1 != AsyncReset2) ||
    (AsyncResetPol1 != AsyncResetPol2) ||
    (SetPriorReset1 == SetPriorReset2))
{

```

gnl fsmread.c

```
fprintf (stderr,
        " ERROR: state variable <%s> cannot be synthesized\n",
        GnlVarName (OutVar));
return (GNL_BAD_FSM_FOR_SYNTHESIS);
}

/* Actually there is no real priority between Set and Reset.      */
/* The priority depends on the dontcare value.                  */
SetPriorReset = 2;

/* Now we create physically a new GNL_SEQUENTIAL_COMPONENT in      */
/* the list of components of the current 'Gnl'.                  */
if (GnlCreateFsmSeqComponent (Gnl, 0, OutVar, StateVar,
                               BddInput1, ListFct,
                               ClockVar1, ClockPol1,
                               AsyncSet1, AsyncSetPol1,
                               AsyncReset1, AsyncResetPol1,
                               SetPriorReset))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

if ((Status = GnlExtractDffSeqCompo (Gnl, OutVar, StateVar, Bdd, Support,
                                      &BddInput, &ClockVar, &ClockPol, &AsyncSet,
                                      &AsyncSetPol, &AsyncReset, &AsyncResetPol,
                                      &SetPriorReset, &Error)))
    return (Status);

if (!Error)
{
    /* Now we create physically a new GNL_SEQUENTIAL_COMPONENT in      */
    /* the list of components of the current 'Gnl'.                  */
    if (GnlCreateFsmSeqComponent (Gnl, 1, OutVar, StateVar, BddInput,
                                   ListFct, ClockVar, ClockPol,
                                   AsyncSet, AsyncSetPol,
                                   AsyncReset, AsyncResetPol,
                                   SetPriorReset))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

if ((Status = GnlExtractLatchSeqCompo (Gnl, OutVar, StateVar, Bdd,
                                       &BddInput, &ClockVar, &ClockPol, &AsyncSet,
                                       &AsyncSetPol, &AsyncReset, &AsyncResetPol,
                                       &SetPriorReset, &Error)))
    return (Status);

if (Error)
{
    fprintf (stderr,
            " ERROR: state variable <%s> cannot be synthesized\n",
            GnlVarName (OutVar));
    return (GNL_BAD_FSM_FOR_SYNTHESIS);
}
```

gnlfsmread.c

```
/* Now we create physically a new GNL_SEQUENTIAL_COMPONENT in */
/* the list of components of the current 'Gnl'. */
if (GnlCreateFsmSeqComponent (Gnl, 0, OutVar, StateVar, BddInput,
                             ListFct, ClockVar, ClockPol,
                             AsyncSet, AsyncSetPol,
                             AsyncReset, AsyncResetPol,
                             SetPriorReset))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlNameIncluded
 */
/*
 * This procedures verifies in Name1 is included in Name2. Actually
 * Name1 is at the end of Name2 if is is the case.
 *      Name2 = ???@Name1
 */
int GnlNameIncluded (Name1, Name2)
    char          *Name1;
    char          *Name2;
{
    int           L1;
    int           L2;
    int           Diff;
    int           i;

    L1 = strlen (Name1);
    L2 = strlen (Name2);
    Diff = L2-L1;

    for (i=0; i<L1; i++)
    {
        if (Name1[i] != Name2[Diff+i])
            return (0);
    }

    if (Name2[Diff-1] != '@')
        return (0);

    return (1);
}

/*
 * GnlCorrespondingOutVar
 */
int GnlCorrespondingOutVar (StateVar, Var)
    GNL_VAR StateVar;
    GNL_VAR Var;
{
    int           S1;
    int           l;
```

gnlfsmread.c

```
sl = strlen (GnlVarName (StateVar));
l = strlen (GnlVarName (Var));

if (Sl == l)
    return (0);

if (Sl > l)
    return (GnlNameIncluded (GnlVarName (Var), GnlVarName (StateVar)));
else
    return (GnlNameIncluded (GnlVarName (StateVar), GnlVarName (Var)));

}

/*-----
/* GnlGetOutVarFromStateVar
/*-----*/
GNL_STATUS GnlGetOutVarFromStateVar (Gnl, StateVar, OutVar)
    GNL             Gnl;
    GNL_VAR        StateVar;
    GNL_VAR        *OutVar;
{
    int            i;
    int            j;
    BLIST          HashTableNames;
    BLIST          BucketI;
    GNL_VAR        VarJ;

HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        if (VarJ == StateVar)
            continue;
        if (GnlCorrespondingOutVar (StateVar, VarJ) )
        {
            *OutVar = VarJ;
            return (GNL_OK);
        }
    }
}
fprintf (stderr,
    "ERROR: did not find corresponding output var of state var <%s>\n",
    GnlVarName (StateVar));
exit (1);

*OutVar = NULL;

return (GNL_OK);
}
```

gnlfsmread.c

```
----- */
/* GnlDecreaseDads */
----- */
void GnlDecreaseDads (Gnl, Node)
    GNL          Gnl;
    GNL_NODE Node;
{
    int           i;
    GNL_VAR     Var;
    GNL_NODE NodeI;
    GNL_FUNCTION Function;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        Function = GnlVarFunction (Var);
        if (!Function)
            return;

        SetGnlVarDads (Var, (int)GnlVarDads (Var)-1);
        NodeI = GnlFunctionOnSet (Function);
        if ((GnlVarDads (Var) == 0) &&
            ((GnlVarDir (Var) == GNL_VAR_LOCAL) ||
             (GnlVarDir (Var) == GNL_VAR_FSM_FUNCTION)))
        {
            GnlFunctionFree (Function);
            GnlResetVarFunction (Var);
            GnlDecreaseDads (Gnl, NodeI);
        }
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        GnlDecreaseDads (Gnl, NodeI);
    }
}

----- */
/* GnlRemoveUnusedVarInGnl */
----- */
/* This procedure removes function variables which are not used. */
----- */
void GnlRemoveUnusedVarInGnl (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_VAR     VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE NodeI;
```

```

/* Resetting the Dads field of each functiona var. */  

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)  

{  

    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  

    SetGnlVarDads (VarI, NULL);  

}  

/* we compute the number of Dads for Variable. */  

GnlGetNumberDadsFromGnl (Gnl);  

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)  

{  

    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  

    FunctionI = GnlVarFunction (VarI);  

    if (!FunctionI)  

        continue;  

    if ((GnlVarDads (VarI) != 0) ||  

        ((GnlVarDir (VarI) != GNL_VAR_LOCAL) &&  

         (GnlVarDir (VarI) != GNL_VAR_FSM_FUNCTION)))  

        continue;  

    NodeI = GnlFunctionOnSet (FunctionI);  

    GnlFunctionFree (FunctionI);  

    GnlResetVarFunction (VarI);  

    GnlDecreaseDads (Gnl, NodeI);  

}  

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)  

{  

    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  

    FunctionI = GnlVarFunction (VarI);  

    if (!FunctionI)  

    {  

        BListDelInsert (GnlFunctions (Gnl), i+1);  

        i--;  

    }
}
  

/*-----*/  

/* GnlReplaceStateVarByOutVarInNode */  

/*-----*/  

void GnlReplaceStateVarByOutVarInNode (Node, StateVar, OutVar)  

{
    GNL_NODE          Node;  

    GNL_VAR           StateVar;  

    GNL_VAR           OutVar;  

{  

    int               i;  

    GNL_NODE         SonI;  

    GNL_VAR          Var;
}

```

gnl fsmread.c

```
if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return ;

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    Var = (GNL_VAR) GnlNodeSons (Node);
    if (Var == StateVar)
        SetGnlNodeSons (Node, (BLIST) OutVar);
    return ;
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
    GnlReplaceStateVarByOutVarInNode (SonI, StateVar, OutVar);
}
}

/*-----*/
/* GnlReplaceStateVarByOutVar */
/*-----*/
void GnlReplaceStateVarByOutVar (Gnl, StateVar, OutVar)
    GNL          Gnl;
    GNL_VAR      StateVar;
    GNL_VAR      OutVar;
{
    int           i;
    GNL_VAR      Vari;
    GNL_FUNCTION FunctionI;
    GNL_NODE     NodeI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR) BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (Vari);
        if (!FunctionI)
            continue;

        NodeI = GnlFunctionOnSet (FunctionI);

        GnlReplaceStateVarByOutVarInNode (NodeI, StateVar, OutVar);
    }
}

/*-----*/
/* GnlPrintBadSequentialExpression */
/*-----*/
GNL_STATUS GnlPrintBadSequentialExpression (Bdd)
    BDD          Bdd;
{
    BLIST        Support;
    int          i;
    int          IndexBddI;
    GNL_VAR      CofVar;
```

gnlfsmread.c

```
BDD           RealBdd;
int          ClockEdge;

ClockEdge = 0;

if (GnlGetSupportOfFsmBdd (Bdd, &Support))
    return (GNL_MEMORY_FULL);

/* First we scan the support in order to remove the master clock of */
/* name GNL_MASTER_CLOCK_EVENT. We get then the Bdd 'RealBdd'. */
for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);
    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (!strcmp (GnlVarName (CofVar), GNL_MASTER_CLOCK_EVENT))
    {
        if (bdd_cofacl (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);
        BListDelInsert (Support, i+1);
        Bdd = RealBdd;
        break;
    }
}

for (i=0; i<BListSize (Support); i++)
{
    IndexBddI = (int)BListElt (Support, i);
    CofVar = GnlGetGnlVarFromBddIndex (IndexBddI);
    if (GnlClockEventVar (CofVar))
    {
        if (bdd_cofacl (Bdd, IndexBddI, &RealBdd))
            return (GNL_MEMORY_FULL);
        BListDelInsert (Support, i+1);
        Bdd = RealBdd;
        i--;
        ClockEdge++;
    }
}

switch (ClockEdge) {
    case 1:
        fprintf (stderr,
"Expression depends on one clock signal. Make sure that\n");
        fprintf (stderr,
"sensitivity list is correctly defined.\n");
        fprintf (stderr, "\n");
        break;

    case 0:
        fprintf (stderr,
"Expecting a Latch inference but some signals are not sensitive\n");
        fprintf (stderr,
"Make sure that sensitivity list is correctly defined.\n");
        break;
}
```

gnlfsmread.c

```
    default:
        fprintf (stderr,
"        Expression depends on several clocks. Make sure that\n");
        fprintf (stderr,
"        sensitivity list is correctly defined.\n");
        break;
    }

    return (GNL_OK);
}

/*-----
/* GnlExtractFsmSequentialCompo
/*-----*/
GNL_STATUS GnlExtractFsmSequentialCompo (Gnl)
    GNL          Gnl;
{
    BLIST      HashListFsmVar;
    BLIST      ListStateVar;
    int         i;
    GNL_VAR    StateVarI;
    BDD_STATUS Status;
    BDD_WS     Ws;
    BDD         BddExpr;
    GNL_STATUS GnlStatus;
    GNL_VAR    OutVarI;
    GNL_VAR    VarI;
    BLIST      HashTableNames;
    BLIST      BucketI;
    int         j;
    GNL_VAR    VarJ;
    BLIST      ListFct;
    BLIST      ListStateOutVar;

    GnlResetDadsInVar (Gnl);

    /* we compute the number of Dads for Variable. */
    /* GnlGetNumberDadsFromGnl (Gnl); */

    HashListFsmVar = GnlHashListFsmVar (Gnl);
    ListStateVar = GnlListStateVar (Gnl);

    /* no state variables */
    if (BListSize (ListStateVar) == 0)
        return (GNL_OK);

    if ((Status = InitBddWorkSpace (50, &Ws)))
        return (GNL_MEMORY_FULL);

    if ((Status = GnlBuildBddOnFsmVarInput (Gnl)))
        return (GNL_MEMORY_FULL);

    /* we identify specific signals */
    if (BListCreate (&ListStateOutVar))
        return (GNL_MEMORY_FULL);
```

gnl fsmread.c

```
for (i=0; i<BListSize (ListStateVar)-1; i = i+2)
{
    StateVarI = (GNL_VAR)BListElt (ListStateVar, i);

    if (GnlGetOutVarFromStateVar (Gnl, StateVarI, &OutVarI))
        return (GNL_MEMORY_FULL);

    if (!GnlVarIsPrimary (StateVarI))
        SetGnlVarDir (StateVarI, GNL_STATE_VAR);

    if (!GnlVarIsPrimary (OutVarI))
        SetGnlVarDir (OutVarI, GNL_OUT_STATE_VAR);

    if (BListAddElt (ListStateOutVar, (int)StateVarI))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListStateOutVar, (int)OutVarI))
        return (GNL_MEMORY_FULL);
}

/* 'ListStateOutVar' is a list of couples (statevar, OutVar) */
for (i=0; i<BListSize (ListStateOutVar)-1; i = i+2)
{
    StateVarI = (GNL_VAR)BListElt (ListStateOutVar, i);
    OutVarI = (GNL_VAR)BListElt (ListStateOutVar, i+1);

    if (GnlVarDriveFuncId (OutVarI) != 1)
    {
        /* There is a driving condition on this signal.          */
        fprintf (stderr,
        " WARNING: driving condition on a stored variable <%s>\n",
                GnlVarName (OutVarI));
        exit (1);
    }

    if (GnlBuildBddFsmExpr (OutVarI, StateVarI, &ListFct, &BddExpr))
        return (GNL_MEMORY_FULL);

    /* The bdd expression of the output is a simple constante */
    if ((BddExpr == bdd_one ()) ||
        (BddExpr == bdd_zero ()))
        continue;

    if ((GnlStatus = GnlExtractSeqCompoFromBdd (Gnl, OutVarI,
                                                StateVarI, BddExpr, ListFct)))
    {
        if (GnlPrintBadSequentialExpression (BddExpr))
            return (GNL_MEMORY_FULL);
        return (GnlStatus);
    }

    GnlResetVarFunction (OutVarI);
}

for (i=0; i<BListSize (ListStateOutVar)-1; i = i+2)
{
    StateVarI = (GNL_VAR)BListElt (ListStateOutVar, i);
```

gnlfsmread.c

```
OutVarI = (GNL_VAR)BListElt (ListStateOutVar, i+1);

/* We replace all 'state var' occurrences by the corresponding      */
/* 'out var'.                                                       */
GnlReplaceStateVarByOutVar (Gnl, StateVarI, OutVarI);

if (!GnlVarIsPrimary (OutVarI))
    SetGnlVarDir (OutVarI, GNL_VAR_LOCAL);
}

BListQuickDelete (&ListStateOutVar);

HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        if (GnlVarDir (VarJ) == GNL_STATE_VAR)
        {
            BListDelInsert (BucketI, j+1);
            j--;
        }
    }
}

FreeBddWorkSpace (Ws);

return (GNL_OK);
}

/*
 * GnlUpdateVarDirInGnl
 */
/*
 * We update the var direction of all the variables
 */
void GnlUpdateVarDirInGnl (Gnl)
    GNL      Gnl;
{
    BLIST      HashTableNames;
    int         i;
    int         j;
    GNL_VAR   VarJ;
    BLIST      BucketI;

HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        if (GnlVarDir (VarJ) == GNL_VAR_LOCAL_DONTCARE)
            SetGnlVarDir (VarJ, GNL_VAR_LOCAL);
    }
}
```

gnl fsmread.c

```
if (GnlVarDir (VarJ) == GNL_VAR_FSM_FUNCTION)
    SetGnlVarDir (VarJ, GNL_VAR_LOCAL);
}
}

/*
 * GnlExtractFsmTriStateCompo
 */
GNL_STATUS GnlExtractFsmTriStateCompo (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR      OutVar;
    GNL_FUNCTION FunctionI;
    GNL_NODE     DriveFuncNode;
    GNL_VAR      InputVar;
    GNL_VAR      EnableVar;
    GNL_FUNCTION NewFunction;
    GNL_NODE     InputFuncNode;
    char         *NewName;
    GNL_TRISTATE_COMPONENT NewTristate;
    BDD          BddInput;
    BDD          BddDrive;
    BDD          NewBddInput;
    BDD_STATUS   Status;
    BDD_WS       Ws;
    int          SelectPol;
    int          InputPol;
    BDD_PTR     BddInputPtr;
    BDD_PTR     BddDrivePtr;

    if ((Status = InitBddWorkSpace (50, &Ws)))
        return (GNL_MEMORY_FULL);

    if ((Status = GnlBuildBddOnFsmVarInput (Gnl)))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        OutVar = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);

        FunctionI = GnlVarFunction (OutVar);
        if (!FunctionI)
            continue;

        if (!GnlFunctionZSet (FunctionI))
            continue;

        /* Defining the input var of the tri-state */
        InputFuncNode = GnlFunctionOnSet (FunctionI);
        if (GnlBuildBddFsmExprOnNode (OutVar, OutVar,
                                      InputFuncNode, 0, 0, NULL, NULL,
                                      NULL, &BddInput))
```

gnlfsmread.c

```
        return (BDD_MEMORY_FULL);

/* Defining the var for the enable of the tri-state */  
DriveFuncNode = GnlFunctionZSet (FunctionI);  
if (GnlBuildBddFsmExprOnNode (OutVar, OutVar,  
                                DriveFuncNode, 0, 0, NULL, NULL,  
                                NULL, &BddDrive))  
    return (BDD_MEMORY_FULL);

#ifndef BUG  
    /* We AND the Input with the selector function to get the new */  
    /* input. */  
    if (bdd_restrict (BddInput, BddDrive, &NewBddInput))  
        return (BDD_MEMORY_FULL);

#else  
    NewBddInput = BddInput;BddInputPtr = (BDD_PTR)GetBddPtrFromBdd (NewBddInput);  
if (NewBddInput != (BDD)BddInputPtr)  
{  
    InputPol = 0;  
    if (GnlCreateInputSeqCompoVar (Gnl, (BDD)BddInputPtr, "d",  
                                   NULL, NULL, &InputVar))  
        return (BDD_MEMORY_FULL);  
}  
else  
{  
    if (GnlCreateInputSeqCompoVar (Gnl, NewBddInput, "d",  
                                   NULL, NULL, &InputVar))  
        return (BDD_MEMORY_FULL);  
}

SelectPol = 1;          /* transparent at high level */  
BddDrivePtr = (BDD_PTR)GetBddPtrFromBdd (BddDrive);  
if (BddDrive != (BDD)BddDrivePtr)  
{  
    SelectPol = 0;  
    if (GnlCreateInputSeqCompoVar (Gnl, (BDD)BddDrivePtr, "en",  
                                   NULL, NULL, &EnableVar))  
        return (BDD_MEMORY_FULL);  
}  
else  
{  
    if (GnlCreateInputSeqCompoVar (Gnl, BddDrive, "en",  
                                   NULL, NULL, &EnableVar))  
        return (BDD_MEMORY_FULL);  
}

if (GnlCreateTristateComponent (&NewTristate))  
    return (GNL_MEMORY_FULL);
```

gnlfsmread.c

```
if (GnlStrCopy (GnlVarName (OutVar), &NewName))
    return (GNL_MEMORY_FULL);

SetGnlTriStateInstName (NewTristate, NewName);

SetGnlTriStateOutput (NewTristate, OutVar);
SetGnlTriStateInput (NewTristate, InputVar);
SetGnlTriStateInputPol (NewTristate, InputPol);
SetGnlTriStateSelect (NewTristate, EnableVar);
SetGnlTriStateSelectPol (NewTristate, SelectPol);

if (!GnlVarIsPrimary (OutVar))
    SetGnlVarDir (OutVar, GNL_VAR_LOCAL_WIRING);
if (!GnlVarIsPrimary (InputVar))
    SetGnlVarDir (InputVar, GNL_VAR_LOCAL_WIRING);
if (!GnlVarIsPrimary (EnableVar))
    SetGnlVarDir (EnableVar, GNL_VAR_LOCAL_WIRING);

if (BListAddElt (GnlComponents (Gnl), (int)NewTristate))
    return (GNL_MEMORY_FULL);

GnlResetVarFunction (OutVar);
}

FreeBddWorkSpace (Ws);

return (GNL_OK);
}

/*-----
/* GnlNameFromSeqOp
*/
char *GnlNameFromSeqOp (Op)
    GNL_SEQUENTIAL_OP    Op;
{

switch (Op) {
    case GNL_DFF:
    case GNL_DFF0:
    case GNL_DFF1:
    case GNL_DFFX:
        return (" Dff ");

    case GNL_LATCH:
    case GNL_LATCH1:
    case GNL_LATCH0:
    case GNL_LATCHX:
        return ("Latch");
}
}

/*-----
/* GnlConflictValueFromOp
*/
char GnlConflictValueFromOp (Op)
    GNL_SEQUENTIAL_OP    Op;
```

```

{
    switch (Op) {
        case GNL_DFF:
        case GNL_LATCH:
            return (' ');
        case GNL_DFF0:
        case GNL_LATCH0:
            return ('R');
        case GNL_DFF1:
        case GNL_LATCH1:
            return ('S');
        case GNL_DFFX:
        case GNL_LATCHX:
            return ('X');
    }
}

/*-----
/* GnlPrintTriStateCompoInfo
/*-----*/
GnlPrintTriStateCompoInfo (TriStateCompo)
    GNL_TRISTATE_COMPONENT      TriStateCompo;
{
    int                  i;
    char                 *OutVarName;

    fprintf (stderr,
             "      ");
    OutVarName = GnlVarName (GnlTriStateOutput (TriStateCompo));
    GnlPrintFormatSignalName (stderr, OutVarName, 20);

    fprintf (stderr, "      TriState      ");

    if (GnlTriStateSelectPol (TriStateCompo))
        fprintf (stderr, "      H      ");
    else
        fprintf (stderr, "      L      ");

    if (GnlTriStateInputPol (TriStateCompo))
        fprintf (stderr, "      H      \n");
    else
        fprintf (stderr, "      L      \n");
}
}

/*-----
/* GnlPrintSeqCompoInfo
/*-----*/
void GnlPrintSeqCompoInfo (SeqCompo, NbDffs, NbResets, NbSets, NbRSs)
    GNL_SEQUENTIAL_COMPONENT      SeqCompo;
    int                  *NbDffs;
    int                  *NbResets;
    int                  *NbSets;
    int                  *NbRSs;
}

```

gnl fsmread.c

```
int i;
char *OutVarName;
char ValueConflict;
GNL_VAR Input;
GNL_NODE Node;
int Value;

*NbDffs = 1;
*NbResets = *NbSets = *NbRSSs = 0;
fprintf (stderr,
        " ");
OutVarName = GnlVarName (GnlSequentialCompoOutput (SeqCompo));
GnlPrintFormatSignalName (stderr, OutVarName, 20);

fprintf (stderr, "%s",
        GnlNameFromSeqOp (GnlSequentialCompoOp (SeqCompo)));

if (GnlVarIsVdd (GnlSequentialCompoClock (SeqCompo)) ||
    GnlVarIsVss (GnlSequentialCompoClock (SeqCompo)))
{
    fprintf (stderr,
            "[%s] ",
            GnlVarName (GnlSequentialCompoClock (SeqCompo)));
}
else if (GnlSequentialCompoClockPol (SeqCompo))
    fprintf (stderr, " H ");
else
    fprintf (stderr, " L ");

if (GnlSequentialCompoReset (SeqCompo))
{
    if (GnlSequentialCompoResetPol (SeqCompo))
        fprintf (stderr, " H ");
    else
        fprintf (stderr, " L ");
    *NbResets = 1;
}
else
    fprintf (stderr, " . ");

if (GnlSequentialCompoSet (SeqCompo))
{
    if (GnlSequentialCompoSetPol (SeqCompo))
        fprintf (stderr, " H ");
    else
        fprintf (stderr, " L ");
    *NbSets = 1;
}
else
    fprintf (stderr, " . ");

if (GnlSequentialCompoReset (SeqCompo) &&
    GnlSequentialCompoSet (SeqCompo))
{
    ValueConflict = GnlConflictValueFromOp (
        GnlSequentialCompoOp (SeqCompo));
}
```

gnlfsmread.c

```
    fprintf (stderr, " %c ", ValueConflict);
    *NbRSSs = 1;
}
else
    fprintf (stderr, " . ");

Input = GnlSequentialCompoInput (SeqCompo);
if (GnlVarFunction (Input))
{
    Node = GnlFunctionOnSet (GnlVarFunction (Input));
    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        Value = (int)GnlNodeSons (Node);
        fprintf (stderr, "--> d = [%d]", Value);
    }
}
fprintf (stderr, "\n");

}

/*-----*/
/* GnlPrintDffInfos */
/*-----*/
void GnlPrintDffInfos (Components, NbDffs, NbResets, NbSets, NbRSSs)
BLIST Components;
int *NbDffs;
int *NbResets;
int *NbSets;
int *NbRSSs;
{
    int i;
    GNL_COMPONENT ComponentI;
    GNL_SEQUENTIAL_COMPONENT SeqCompoI;
    int NbDff;
    int NbReset;
    int NbSet;
    int NbRS;

*NbDffs = *NbResets = *NbSets = *NbRSSs = 0;
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
        continue;
    SeqCompoI = (GNL_SEQUENTIAL_COMPONENT)ComponentI;

    if ((GnlSequentialCompoOp (SeqCompoI) != GNL_DFF) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_DFF0) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_DFF1) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_DFFX))
        continue;

    GnlPrintSeqCompoInfo ((GNL_SEQUENTIAL_COMPONENT)ComponentI,
                          &NbDff, &NbReset, &NbSet, &NbRS);
    *NbDffs += NbDff;
    *NbResets += NbReset;
```

gnlfsmread.c

```
*NbSets += NbSet;
*NbRSs += NbRS;
}
}

/*
/* GnlPrintLatchInfos
*/
void GnlPrintLatchInfos (Components, NbDffs, NbResets, NbSets, NbRSs)
    BLIST      Components;
    int        *NbDffs;
    int        *NbResets;
    int        *NbSets;
    int        *NbRSs;
{
    int          i;
    GNL_COMPONENT ComponentI;
    GNL_SEQUENTIAL_COMPONENT SeqCompoI;
    int          NbDff;
    int          NbReset;
    int          NbSet;
    int          NbRS;

*NbDffs = *NbResets = *NbSets = *NbRSs = 0;
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
        continue;
    SeqCompoI = (GNL_SEQUENTIAL_COMPONENT)ComponentI;

    if ((GnlSequentialCompoOp (SeqCompoI) != GNL_LATCH) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_LATCH0) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_LATCH1) &&
        (GnlSequentialCompoOp (SeqCompoI) != GNL_LATCHX))
        continue;

    GnlPrintSeqCompoInfo ((GNL_SEQUENTIAL_COMPONENT)ComponentI,
                          &NbDff, &NbReset, &NbSet, &NbRS);

    *NbDffs += NbDff;
    *NbResets += NbReset;
    *NbSets += NbSet;
    *NbRSs += NbRS;
}
}

/*
/* GnlPrintTriStateInfos
*/
void GnlPrintTriStateInfos (Components)
    BLIST      Components;
{
    int          i;
    GNL_COMPONENT ComponentI;
    int          NbTriState;
```

gnlfsmread.c

```
fprintf (stderr, "\n");
fprintf (stderr,
        "      Signal           Type           Select       Input \n");
fprintf (stderr,
        "      -----\n");
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_TRISTATE_COMPO)
        continue;

    GnlPrintTriStateCompoInfo ((GNL_TRISTATE_COMPONENT)ComponentI);
}
fprintf (stderr,
        "      -----\\n");
}

/*-----*/
/* GnlPrintComponentInfos */
/*-----*/
void GnlPrintComponentInfos (Gnl)
{
    GNL      Gnl;
{
    int          i;
    BLIST      Components;
    GNL_COMPONENT ComponentI;
    int          NbSeq;
    int          NbTriState;
    int          NbDffs1;
    int          NbReset1;
    int          NbSet1;
    int          NbRS1;
    int          NbDffs2;
    int          NbReset2;
    int          NbSet2;
    int          NbRS2;

    fprintf (stderr, "\n ======");
    for (i=0; i<strlen (GnlName (Gnl))-1; i++)
        fprintf (stderr, "=");
    fprintf (stderr, " ");
    fprintf (stderr, "\n Components of module [%s] \n", GnlName (Gnl));
    fprintf (stderr, " ======");
    for (i=0; i<strlen (GnlName (Gnl))-1; i++)
        fprintf (stderr, "=");
    fprintf (stderr, " \n");

    Components = GnlComponents (Gnl);
    if (!Components || (BListSize (Components) == 0))
    {
        fprintf (stderr, " None.\n\n");
        return;
    }
}
```

```

}

NbSeq = 0;
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
        continue;
    NbSeq++;
}

if (NbSeq)
{
    fprintf (stderr, "\n");
    fprintf (stderr,
    "      Signal           Type     Clock     Reset     Set      R/S \n");
    fprintf (stderr,
    "-----\n");
    GnlPrintDffInfos (Components, &NbDffs1, &NbReset1, &NbSet1,
                      &NbRS1);

    GnlPrintLatchInfos (Components, &NbDffs2, &NbReset2, &NbSet2,
                        &NbRS2);
    fprintf (stderr,
    "-----\n");

    if (NbDffs1)
        fprintf (stderr,
    "      Nb. Dffs       %6d      %6d %6d\n",
    NbDffs1, NbReset1, NbSet1, NbRS1);
    if (NbDffs2)
        fprintf (stderr,
    "      Nb. Latches    %6d      %6d %6d\n",
    NbDffs2, NbReset2, NbSet2, NbRS2);
    fprintf (stderr,
    "-----\n");
}

NbTriState = 0;
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_TRISTATE_COMPO)
        continue;
    NbTriState++;
}

if (NbTriState)
{
    GnlPrintTriStateInfos (Components);
    fprintf (stderr,
    "      Nb. TriStates      %6d\n", NbTriState);
    fprintf (stderr,
    "-----\n");
}

```

```

if (!NbSeq && !NbTriState)
    fprintf (stderr, " Only user components.\n\n");
else
    fprintf (stderr, "\n");
}

/*-----
/* GnlAddVarInFsmCompoInterface
/*-----
/* Name of the var is of the form: instance_name.var_name      */
/* we extract then: instance_name and var_name.                */
/* From 'instance_name' we look for the components of same name and   */
/* updates its interface.                                         */
/*-----*/
GNL_STATUS GnlAddVarInFsmCompoInterface (Gnl, Var)
    GNL           Gnl;
    GNL_VAR      Var;
{
    char          *Name;
    int            Index;
    int            i;
    int            L;
    GNL_USER_COMPONENT TheComponent;
    GNL_USER_COMPONENT ComponentI;
    char          *VarName;
    GNL_FUNCTION   Function;
    GNL_NODE       Node;
    GNL_ASSOC      NewAssoc;
    GNL_VAR        LeftVar;

    Name = GnlVarName (Var);

    Index = 0;
    while (Name[Index] != '.') Index++; /* there is a '.' for sure */
    Name[Index] = '\0';

    TheComponent = NULL;
    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        /* Component is here a USER component                      */
        ComponentI = (GNL_USER_COMPONENT)
                     BlistElt (GnlComponents (Gnl), i);
        if (!strcmp (GnlUserComponentInstName (ComponentI), Name))
        {
            TheComponent = ComponentI;
            break;
        }
    }

    Name[Index] = '.';

    /* 'TheComponent' is the component                           */
    if (!TheComponent)

```

```

{
    fprintf (stderr,
              " ERROR: cannot find FSM instance <%s>\n", Name);
    return (GNL_OK);
}

L = strlen (Name);
if ((VarName = (char*)
        calloc (L-Index+1, sizeof (char))) == NULL)
    return (GNL_MEMORY_FULL);

Index++;
i = Index;
while (i < L)
{
    VarName[i-Index] = Name[i];
    i++;
}
VarName[i-Index] = '\0';

/* If the var is a black box inout then there is a loop. The Inout */
/* var must recursively leads to an other inout var. It can be of */
the*/
/* form: a1 = i1.a AND i2.a and */          */
/*       i1.a = a1 */                         */
/*       i2.a = a1 */                         */
/* */                                         */
/* Ex VHDL leading this situation: */          */
/* entity hierio is */                      */
/*     port (b1, b2: in bit_vector (0 to 1); a1, a2, c1, c2, d1, d2: */ */
/*             inout std_logic; z1, z2: out bit_vector (0 to 1)); */ */
/* end hierio; */                           */
/* */                                         */
/* architecture arch1 of hierio is */          */
/* */                                         */
/* component hierio2 */                      */
/*     port (b: in bit_vector (0 to 1); */      */
/*             a, c, d: inout std_logic; z : out bit_vector (0 to 1)); */ */
/* end component; */                         */
/* component hierio3 */                      */
/*     port (b: in bit_vector (0 to 1); */      */
/*             a, c, d: in bit; z : out bit_vector (0 to 1)); */ */
/* end component; */                         */
/* */                                         */
/* begin */                                */
/* */                                         */
/* i1: hierio2 port map (a => a1, b => b2, c => c2, d => d2, z=>z2); */
/* i2: hierio2 port map (a => a1, b => b1, c => c1, d => d1, z=>z1); */
/* */                                         */
/* end arch1; */                            */

/* There is a loop we must cut. We create a predefined component */
/* GNL_BIDIR_COMPONENT for each equations i1.a = a1 and i2.a = a1 */
if (GnlVarDir (Var) == GNL_VAR_FSM_BLACK_BOX_INOUT)
{
    Function = GnlVarFunction (Var);
    Node = GnlFunctionOnSet (Function);
}

```

gnlfsmread.c

```
/* we should never enter this if condition. */  
if (GnlNodeOp (Node) != GNL_VARIABLE)  
{  
    fprintf (stderr,  
    " ERROR: Mapit cannot handle such INOUT usage for signal <%s>\n",  
            GnlVarName (Var));  
    exit (1);  
}  
/* we remove the equation and create a GNL_BIDIR_COMPONENT */  
free (Function);  
SetGnlVarFunction (Var, NULL);  
GnlRemoveFunctionFromGnl (Gnl, Var);  
  
LeftVar = (GNL_VAR)GnlNodeSons (Node);  
}  
  
if (GnlCreateAssoc (&NewAssoc))  
    return (GNL_MEMORY_FULL);  
  
if (!GnlVarIsPrimary (Var))  
    SetGnlVarDir (Var, GNL_VAR_LOCAL_WIRING);  
  
SetGnlAssocFormalPort (NewAssoc, VarName);  
SetGnlAssocActualPort (NewAssoc, Var);  
  
if (BListAddElt (GnlUserComponentInterface (TheComponent),  
                 (int)NewAssoc))  
    return (GNL_MEMORY_FULL);  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlUpdateUserCompoInterface */  
/*-----*/  
GNL_STATUS GnlUpdateUserCompoInterface (Gnl)  
    GNL  
    Gnl;  
{  
    BLIST      HashTableNames;  
    int         i;  
    BLIST      BucketI;  
    int         j;  
    GNL_VAR    VarJ;  
    GNL_NODE   Node;  
    GNL_VAR    Var;  
  
    HashTableNames = GnlHashNames (Gnl);  
    for (i=0; i<BListSize (HashTableNames); i++)  
    {  
        BucketI = (BLIST)BListElt (HashTableNames, i);  
        for (j=0; j<BListSize (BucketI); j++)  
        {  
            VarJ = (GNL_VAR)BListElt (BucketI, j);  
            if ((GnlVarDir (VarJ) == GNL_VAR_FSM_BLACK_BOX) ||  
                (GnlVarDir (VarJ) == GNL_VAR_FSM_BLACK_BOX_INOUT))
```

gnlfsmread.c

```
{  
    if (!GnlVarRangeUndefined (VarJ))  
    {  
        GnlFreeVar (VarJ);  
        BListDelInsert (BucketI, j+1);  
        j--;  
        continue;  
    }  
    if (GnlAddVarInFsmCompoInterface (Gnl, VarJ))  
        return (GNL_MEMORY_FULL);  
    continue;  
}  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlFileNameIsGeneric */  
/*-----*/  
/* A file corresponding to the generic elaboration of a Gnl has a post */  
/* fix number 'index' generated by FSMC of the form: */  
/* entity_name(archi).fsm.index */  
/* If there is such a number then the function returns 1 and 'Index' */  
/* will point on string "index". */  
/*-----*/  
int GnlFileNameIsGeneric (FileName, Index)  
{  
    char *FileName;  
    char **Index;  
    int i;  
    int L;  
  
    *Index = NULL;  
  
    i = 0;  
    L = strlen (FileName);  
    while (i < L)  
    {  
        if (FileName[i] == '.')  
            break;  
        i++;  
    }  
    if (i == L)  
        return (0);  
  
    i++;  
    while (i < L) /* actually we pass 'fsm' */  
    {  
        if (FileName[i] == '.')  
            break;  
        i++;  
    }  
    if (i == L)
```

gnlfsmread.c

```
    return (0);

/* It is then the case of a generic fsm file. */  
*Index = FileName+i;  
  
    return (1);  
  
}  
  
/*-----*/  
/* GnlAnalyzeFsmFiles */  
/*-----*/  
GNL_STATUS GnlAnalyzeFsmFiles (Gnl)  
    GNL          Gnl;  
{  
    int          Status;  
    BLIST        Components;  
    int          i;  
    GNL_USER_COMPONENT ComponentI;  
    int          Exists;  
    int          j;  
    GNL          GnlJ;  
    char         *FileName;  
    FILE         *GnlFile;  
    char         FileNameWithDir[528];  
    char         *GenIndex;  
    char         *GnlName;  
    char         *NewGnlName;  
  
    Components = GnlComponents (Gnl);  
  
    for (i=0; i<BListSize (Components); i++)  
    {  
        ComponentI = (GNL_USER_COMPONENT)BListElt (Components, i);  
  
        /* Gnl definition of 'ComponentI' does not exist */  
        Exists = 0;  
        for (j=0; j<BListSize (G_ListOfGnls); j++)  
        {  
            GnlJ = (GNL)BListElt (G_ListOfGnls, j);  
            if (!strcmp (GnlName (GnlJ),  
                         GnlUserComponentName (ComponentI)))  
            {  
                Exists = 1;  
                break;  
            }  
        }  
  
        if (Exists)  
            continue;  
  
        FileName = (char*)GnlUserComponentHook (ComponentI);  
  
        if (!GnlEnvFsmDir())  
            sprintf (FileNameWithDir, "%s", FileName);  
        else
```

gnlfsmread.c

```
sprintf (FileNameWithDir, "%s/%s", GnlEnvFsmDir(),
         FileName);

if ((GnlFile = freopen (FileNameWithDir , "r", stdin)) == NULL)
{
    fprintf (stderr,
             " WARNING: File '%s' not found:\n", FileNameWithDir);
    fprintf (stderr,
             " [%s] considered as black box\n",
             GnlUserComponentName (ComponentI));
    continue;
}

/* We analyze the sub-fsm. 'G_ListOfGnls' may be modified */
fprintf (stderr, " Analyzing sub-fsm '%s'...\n",
         FileNameWithDir);
if ((Status = yyparse ()))
{
    fclose (GnlFile);
    return (GNL_FSM_PARSE_ERROR);
}
fclose (GnlFile);

/* We modify the name of the Gnl if it corresponds to a generic */
/* fsm file. */
if (GnlFileNameIsGeneric (FileName, &GenIndex))
{
    GnlName = GnlName (G_CurrentGnl);
    if (GnlStrAppendStrCopy (GnlName, GenIndex, &NewGnlName))
        return (GNL_MEMORY_FULL);
    free (GnlName);
    SetGnlName (G_CurrentGnl, NewGnlName);
}

return (GNL_OK);
}

/*
 * GnlSubstituteFsmLocalVariableRec
 */
void GnlSubstituteFsmLocalVariableRec (Node, NewNode)
    GNL_NODE Node;
    GNL_NODE *NewNode;
{
    int i;
    GNL_NODE OnSet;
    GNL_NODE NewOnSet;
    GNL_NODE NewSonI;
    GNL_NODE SonI;
    GNL_FUNCTION Function;
    GNL_VAR Var;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
```

gnl fsmread.c

```
{  
    *NewNode = Node;  
    return;  
}  
  
if (GnlNodeOp (Node) == GNL_VARIABLE)  
{  
    Var = (GNL_VAR) GnlNodeSons (Node);  
    Function = GnlVarFunction (Var);  
    if ((GnlVarDir (Var) != GNL_VAR_FSM_FUNCTION) ||  
        !Function)  
    {  
        *NewNode = Node;  
        return;  
    }  
  
    if (GnlVarDads (Var) != (BLIST) 1)  
    {  
        *NewNode = Node;  
        return;  
    }  
  
    OnSet = GnlFunctionOnSet (Function);  
  
    GnlSubstituteFsmLocalVariableRec (OnSet, &NewOnSet);  
  
    SetGnlFunctionOnSet (Function, NewOnSet);  
  
    *NewNode = NewOnSet;  
    return;  
}  
  
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
{  
    SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);  
    GnlSubstituteFsmLocalVariableRec (SonI, &NewSonI);  
    BListElt (GnlNodeSons (Node), i) = (int) NewSonI;  
}  
  
*NewNode = Node;  
}  
  
/*-----*/  
/* GnlSubstituteFsmLocalVariable */  
/*-----*/  
/* Procedure which removes all the 'f$<index>' fsm variables which are */  
/* not user variables. */  
/*-----*/  
void GnlSubstituteFsmLocalVariable (Gnl)  
{  
    GNL          Gnl;  
    int          i;  
    GNL_VAR      VarI;  
    GNL_FUNCTION Function;  
    GNL_NODE    OnSet;  
    GNL_NODE    NewNode;
```

gnlfsmread.c

```
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    if (!GnlVarFunction (VarI))
        continue;

    /* This is a '$f<index>' variable that we can remove from the      */
    /* equations.                                                       */
    if ((GnlVarDir (VarI) == GNL_VAR_FSM_FUNCTION) &&
        (GnlVarDads (VarI) == (BLIST)1))
    {
        BListDelInsert (GnlFunctions (Gnl), i+1);
        i--;
    }

    Function = GnlVarFunction (VarI);
    OnSet = GnlFunctionOnSet (Function);

    GnlSubstituteFsmLocalVariableRec (OnSet, &NewNode);

    SetGnlFunctionOnSet (Function, NewNode);
}

return;
}

/*-----
/* GnlDumpEquationFileForNova
/*-----*/
GNL_STATUS GnlDumpEquationFileForNova (Gnl)
    GNL       Gnl;
{
    FILE           *OutFile;

    /* we sort the functions from the deepest to the highest ones.          */
    GnlResetDadsInVar (Gnl);

    /* we compute the number of Dads for Variable.                         */
    GnlGetNumberDadsFromGnl (Gnl);

    /* We substitute all the '$f<index>' local variables which are not   */
    /* user defined variables so that the outputs uses only user-defined */
    /* variables.                                                       */
    GnlSubstituteFsmLocalVariable (Gnl);

    if ((OutFile = fopen (GnlEnvOutput(), "w")) == NULL)
    {
        fprintf (stderr, " ERROR: Cannot Open Output File '%s'\n",
                 GnlEnvOutput());
        return (GNL_CANNOT_OPEN_OUTFILE);
    }

    GnlPrintGnl (OutFile, Gnl);
```

gnlfsmread.c

```
fclose (OutFile);

fprintf (stderr, " File '%s' has been generated !\n",
        GnlEnvOutput ());

return (GNL_OK);
}

/*
 * GnlFsmRead
 */
/*
 * Main procedure which reads a '.fsm' file and returns a list of Gnls
 */
GNL_STATUS GnlFsmRead (FileName, ListGnls)
    char          *FileName;
    BLIST        *ListGnls;
{
    FILE          *GnlFile;
    char          *CopyName;
    int           Status;
    int           i;
    GNL           GnlI;
    GNL_STATUS   GnlStatus;
    char          FileNameWithDir[528];
    char          *GnlName;
    char          *GenIndex;
    char          *NewGnlName;

    if (!GnlEnvFsmDir())
        sprintf (FileNameWithDir, "%s", FileName);
    else
        sprintf (FileNameWithDir, "%s/%s", GnlEnvFsmDir(),
                 FileName);

    fprintf (stderr, "\n Reading FSM file [%s] ... \n",
            FileNameWithDir);

    if ((GnlFile = freopen (FileNameWithDir , "r", stdin)) == NULL)
    {
        fprintf (stderr, " ERROR: cannot find file '%s'\n",
                 FileNameWithDir);
        return (GNL_CANNOT_OPEN_INPUTFILE);
    }

    if ((Status = yyparse ()))
    {
        fclose (GnlFile);
        return (GNL_FSM_PARSE_ERROR);
    }
    fclose (GnlFile);

    /* We modify the name of the Gnl if it corresponds to a generic      */
    /* fsm file.                                                       */
    if (GnlFileNameIsGeneric (FileName, &GenIndex))
    {
```

gnlfsmread.c

```
GnlName = GnlName (G_CurrentGnl);
if (GnlStrAppendStrCopy (GnlName, GenIndex, &NewGnlName))
    return (GNL_MEMORY_FULL);
free (GnlName);
SetGnlName (G_CurrentGnl, NewGnlName);
}

fprintf (stderr, "\n FSM File Analyzed [%d] !\n\n",
G_NbBddNodes);

for (i=0; i<BListSize (G_ListOfGnls); i++)
{
    GnLI = (GNL)BListElt (G_ListOfGnls, i);

    if (GnlStrCopy (FileName, &CopyName))
        return (GNL_MEMORY_FULL);

    SetGnlSourceFileName (GnLI, CopyName);

    /* we analyze the sub-fsm files from the components of 'GnLI' */
    /* There is a side-effect on 'G_ListOfGnls'. */
    if ((GnlStatus = GnlAnalyzeFsmFiles (GnLI)))
        return (GnlStatus);

    /* Update Interface of user components */
    if ((GnlStatus = GnlUpdateUserCompoInterface (GnLI)))
        return (GnlStatus);

    if (GnlEnvMode() == GNL_MODE_TRANSLATE)
        GnlPrintGnl (stderr, GnLI);

    if ((GnlStatus = GnlExtractFsmSequentialCompo (GnLI)))
        return (GnlStatus);

    if ((GnlStatus = GnlExtractFsmTriStateCompo (GnLI)))
        return (GnlStatus);

    /* We remove the unused functions from the Gnl. */
    GnlRemoveUnusedVarInGnl (GnLI);

    /* We update the var direction of all the variables */
    GnlUpdateVarDirInGnl (GnLI);

    GnlPrintComponentInfos (GnLI);
}

*ListGnls = G_ListOfGnls;

return (GNL_OK);
}

/*-----
/* GnlSetLocationInVar
/*-----
/* This procedure updates the field 'Location' of GNL_VAR 'Var' by
/* analyzing the list 'ListLoc' which is a list of couples: */
-----*/
```

gnl fsmread.c

```
/*      (id1, id2)  --> Id1 = # source file, Id2 = line number in file      */
/* We create a string corresponding to this information because it is      */
/* more compact for file with number of lines ~ 999999 (6 digits).      */
/* ex: list = {1, 1233, 1, 1433, 2, 2333, 1, 6777, 3, 99999} gives the      */
/* string: '1233 1433 6777 :2333 :99999'.                                */
/*-----*/
GNL_STATUS GnlSetLocationInVar (Var, ListLoc)
    GNL_VAR Var;
    BLIST ListLoc;
{
    int N;
    char LineString[128];
    char *LocString;
    int LocStringI;
    char *CompactLocString;
    int MaxFile;
    int i;
    int j;
    int k;
    int FileJ;
    int LineJ;

    if (BListSize (ListLoc) == 0)
        return (GNL_OK);

    N = BListSize (ListLoc)/2;

    if ((LocString = (char*)calloc (N*64, sizeof(char))) == NULL)
        return (GNL_MEMORY_FULL);

    MaxFile = 1;
    for (i=0; i<BListSize (ListLoc)-1; i = i+2)
    {
        if (MaxFile > (int)BListElt (ListLoc, i))
            MaxFile = (int)BListElt (ListLoc, i);
    }

    LocStringI = 0;
    i = 1;
    while (1)
    {
        for (j=0; j<BListSize (ListLoc)-1; j = j+2)
        {
            FileJ = (int)BListElt (ListLoc, j);

            if (FileJ != i)
                continue;

            LineJ = (int)BListElt (ListLoc, j+1);
            sprintf (LineString, "%d", LineJ);
            for (k=0; k<strlen (LineString); k++)
            {
                LocString[LocStringI] = LineString[k];
                LocStringI++;
            }
            LocString[LocStringI] = ' ';
        }
    }
}
```

gnlfsmread.c

```
        LocStringI++;
    }

    if (i == MaxFile)
        break;

    LocString[LocStringI] = ':';
    LocStringI++;

    i++;
}

LocString[LocStringI] = '\0';

if ((CompactLocString = (char*)
    calloc (strlen(LocString)+1, sizeof(char))) == NULL)
    return (GNL_MEMORY_FULL);

sprintf (CompactLocString, "%s", LocString);

free (LocString);

SetGnlVarLocation (Var, CompactLocString);

return (GNL_OK);
}

/*
/* GnlUpperToLowerChar
*/
char GnlUpperToLowerChar (c)
{
    char      c;
{
    if (c < 65)
        return (c);

    if (c > 90)
        return (c);

    c = c+32;

    return (c);
}

/*
/* GnlUpperToLowerCase
*/
GnlUpperToLowerCase (Name)
{
    char          *Name;
{
    int           L;
    int           i;

    L = strlen (Name);

    for (i=0; i<L; i++)
```

gnlfsmread.c

```
Name[i] = GnlUpperToLowerChar (Name[i]);
}

/*-----
/* GnlAnalyzeInstanceString
/*-----
/* This procedure analyzes a string of the form :
/*     "<instance_name> <library:entity(arch)> <file_name>" */
/*     'InstanceName' becomes: <instance_name>
/*     'GnlName' becomes: <entity-arch>
/*     'FileName' becomes: <file_name-arch>
/*-----
GNL_STATUS GnlAnalyzeInstanceString (InstanceString, InstanceName,
                                     GnlName, FileName)
{
    char      *InstanceString;
    char      **InstanceName;
    char      **GnlName;
    char      **FileName;

    {
        char          TempString[528];
        int           i;
        int           Index;
        char          *GenIndex;
        char          *NewGnlName;

        i=0;
        while (InstanceString[i] != ' ')
        {
            TempString[i] = InstanceString[i];
            i++;
        }
        TempString[i] = '\0';

        if ((*InstanceName = (char*)calloc (strlen (TempString)+1,
                                         sizeof (char))) == NULL)
            return (GNL_MEMORY_FULL);
        sprintf (*InstanceName, "%s", TempString);

        while (InstanceString[i] != ':') i++;
        i++;
        Index = i;

        while (InstanceString[i] != ')')
        {
            if (InstanceString[i] == '(')
                TempString[i-Index] = '-';
            else
                TempString[i-Index] = InstanceString[i];
            i++;
        }
        TempString[i-Index] = '\0';

        if ((*GnlName = (char*)calloc (strlen (TempString)+1,
```

gnlfsmread.c

```
        sizeof (char))) == NULL)
    return (GNL_MEMORY_FULL);
sprintf (*GnlName, "%s", TempString);
GnlUpperToLowerCase (*GnlName);

i++;           /* skip ' ')
i++;           /* skip ' '
Index = i;

while (InstanceString[i] != '\0')
{
    TempString[i-Index] = InstanceString[i];
    i++;
}
TempString[i-Index] = '\0';

if ((*FileName = (char*)calloc (strlen (TempString)+1,
                               sizeof (char))) == NULL)
    return (GNL_MEMORY_FULL);
sprintf (*FileName, "%s", TempString);

/* If the fsm file is a generic fsm file 'ent(arch).fsm.index' then */
/* we change the name 'GnlName' of the Gnl by appending 'index'      */
if (GnlFileNameIsGeneric (*FileName, &GenIndex))
{
    if (GnlStrAppendStrCopy (*GnlName, GenIndex, &NewGnlName))
        return (GNL_MEMORY_FULL);
    free (*GnlName);
    *GnlName = NewGnlName;
}

return (GNL_OK);
}

/*
 * GnlCreateFsmComponents
 */
/*
 * This procedure analyzes the list of instances (string of triplets)
 * and creates the corresponding USER components and store them in the
 * current 'Gnl'.
 */
GNL_STATUS GnlCreateFsmComponents (Gnl, ListInstances)
    GNL          Gnl;
    BLIST       ListInstances;
{
    int           i;
    BLIST       Components;
    char          *InstanceStringI;
    GNL_USER_COMPONENT  NewCompo;
    char          *InstanceName;
    char          *GnlName;
    char          *FileName;
    BLIST       NewList;
```

gnl fsmread.c

```
if (!GnlComponents (Gnl))
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlComponents (Gnl, NewList);
}

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (ListInstances); i++)
{
    InstanceStringI = (char*)BListElt (ListInstances, i);

    if (GnlAnalyzeInstanceString (InstanceStringI,
                                  &InstanceName, &GnlName, &FileName))
        return (GNL_MEMORY_FULL);

    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);

    if (GnlCreateUserComponent (GnlName, InstanceName, NULL, NewList,
                               &NewCompo))
        return (GNL_MEMORY_FULL);

    /* We attach to the hook of this component the file where we can*/
    /* find its definition (we will parse it too).                  */
    SetGnlUserComponentHook (NewCompo, FileName);

    if (BListAddElt (Components, (int)NewCompo))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}
/*-----*/
```

702890 v1

12/28/00
JESSOP LIBRARY
752304
PRO

APPENDIX D

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

gnlhier.c

```

/*
 *-----*/
/*      File:          gnlhier.c
 *      Version:        1.1
 *      Modifications: -
 *      Documentation: -
 */
/*      Description:          */
/*-----*/
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnloption.h"

#include "blist.e"

/*
 *-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV    G_GnlEnv;

/*
 *-----*/
/* GnlGetGnlFromUserCompoName */
/*-----*/
/* returns the Gnl in list 'ListGnls' which name correspond to 'GnlName' */
/*-----*/
GNL GnlGetGnlFromUserCompoName (ListGnls, GnlName)
    BLIST    ListGnls;
    char     *GnlName;
{
    int         i;
    GNL         GnlI;

    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnlI = (GNL)BListElt (ListGnls, i);
        if (!strcmp (GnlName (GnlI), GnlName))
            return (GnlI);
    }
    return (NULL);
}

/*
 *-----*/
/* SortTopLevelModules */
/*-----*/
GNL_STATUS SortTopLevelModules (ListGnls)
    BLIST    ListGnls;

```

gnlhier.c

```
{  
    int i;  
    int j;  
    GNL GnLI;  
    BLIST Components;  
    GNL_COMPONENT ComponentJ;  
    GNL_USER_COMPONENT UserCompoJ;  
    GNL GnlCompo;  
    BLIST ListAux;  
  
    if (BListCreate (&ListAux))  
        return (GNL_MEMORY_FULL);  
  
    for (i=0; i<BListSize (ListGnls); i++)  
    {  
        GnLI = (GNL)BListElt (ListGnls, i);  
        Components = GnlComponents (GnLI);  
        if (!Components)  
            continue;  
  
        for (j=0; j<BListSize (Components); j++)  
        {  
            ComponentJ = (GNL_COMPONENT)BListElt (Components, j);  
            if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)  
                continue;  
  
            UserCompoJ = (GNL_USER_COMPONENT)ComponentJ;  
            GnlCompo = GnlGetGnlFromUserCompoName (ListGnls,  
                                                    GnlUserComponentName (UserCompoJ));  
            if (!GnlCompo)  
                continue;  
            SetGnlRefCount (GnlCompo, GnlRefCount (GnlCompo)+1);  
        }  
    }  
  
    /* We remove the non-top level modules from 'ListGnls'. Only top */  
    /* level module are kept in 'ListGnls'. */  
    for (i=0; i<BListSize (ListGnls); i++)  
    {  
        GnLI = (GNL)BListElt (ListGnls, i);  
  
        /* If the Gnl is used then it is not a top level module. */  
        if (GnlRefCount (GnLI))  
        {  
            if (BListAddElt (ListAux, (int)GnLI))  
                return (GNL_MEMORY_FULL);  
            BListDelInsert (ListGnls, i+1);  
            i--;  
        }  
    }  
  
    /* Adding the non-top level modules after the op ones in the list */  
    /* 'ListGnls'. */  
    for (i=0; i<BListSize (ListAux); i++)  
    {  
        GnLI = (GNL)BListElt (ListAux, i);
```

gnlhier.c

```
if (BListAddElt (ListGnls, (int)GnlI))
    return (GNL_MEMORY_FULL);
}

BListQuickDelete (&ListAux);

return (GNL_OK);
}

/*-----
/* GnlCheckSamePortWidth                               */
/*-----*/
/* Verifies that the formal port and the actual port have the same width*/
/* The actual port can be a GNL_NODE with op. GNL_CONCAT.           */
/*-----*/
int GnlCheckSamePortWidth (Formal, Actual)
    GNL_VAR Formal;
    GNL_VAR Actual;
{
    BLIST Sons;

    if (GnlVarIsVar (Actual))
        return (GnlVarRangeSize (Formal) == GnlVarRangeSize (Actual));

    Sons = GnlNodeSons ((GNL_NODE)Actual);

    return (GnlVarRangeSize (Formal) == (BListSize (Sons)));
}

/*-----
/* GnlCheckCorrectPortDir                            */
/*-----*/
int GnlCheckCorrectPortDir (Formal, Actual)
    GNL_VAR Formal;
    GNL_VAR Actual;
{
    BLIST Sons;
    int i;
    GNL_VAR SonI;

    if (GnlVarIsVar (Actual))
    {
        if ((GnlVarDir (Actual) == GNL_VAR_INPUT) &&
            (GnlVarDir (Formal) == GNL_VAR_OUTPUT))
        {
            fprintf (stderr,
" ERROR: conflict directions between actual <%s> and formal <%s>\n",
                GnlVarName (Actual), GnlVarName (Formal));
            exit (1);
        }
        return (1);
    }

    Sons = GnlNodeSons ((GNL_NODE)Actual);
```

gnlhier.c

```
for (i=0; i<BListSize (Sons); i++)
{
    SonI = (GNL_VAR)BListElt (Sons, i);
    if (!GnlCheckCorrectPortDir (Formal, SonI))
        return (0);
}

return (1);
}

/*-----
/* GnlCheckNUpdateComponentInterface */ */
/*-----
/* This procedure replaces each 'GnlAssocFormalPort' in the associations*/
/* in the interface of 'UserCompo' by the corresponding GNL_VAR in the */
/* corresponding 'Gnl'. */ */
/* At the same time, checks are performed: */
/*     - verify that the formal var exists in the Gnl */
/*     - verify that both formal and actual ports have the same*/
/*         widths. */ */
/*-----*/
GNL_STATUS GnlCheckNUpdateComponentInterface (UserCompo, Gnl)
    GNL_USER_COMPONENT    UserCompo;
    GNL                  Gnl;
{
    BLIST                Interface;
    int                  i;
    GNL_ASSOC             AssocI;
    char                 *FormalPort;
    GNL_VAR               Var;

    /* Already processed. Formal Objects have been already substituted */
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_VAR)
        return (GNL_OK);

    Interface = GnlUserComponentInterface (UserCompo);
    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if (!GnlAssocFormalPort (AssocI))
        {
            fprintf (stderr,
                    " ERROR (l.%d): formal signal #%" GNL_VAR_TYPE "%d missing in [%s]\n",
                    GnlUserComponentLineNumber (UserCompo), i+1,
                    GnlName (Gnl));
            return (GNL_BAD_INSTANCE_INTERFACE);
        }

        FormalPort = (char*)GnlAssocFormalPort (AssocI);
        if (GnlGetVarFromName (Gnl, FormalPort, &Var))
        {
            fprintf (stderr,
                    " ERROR (l.%d): cannot find formal signal <%s> in [%s]\n",
                    GnlUserComponentLineNumber (UserCompo),
                    FormalPort,
```

```

        GnlName(Gnl));
    if (GNL_VAR_NOT_EXISTS)
        return (GNL_BAD_INSTANCE_INTERFACE);
    return (GNL_MEMORY_FULL);
}

/* We free the physical name of 'FormalPort' and replace it */
/* directly by its corresponding var in 'Gnl'.                  */
free (FormalPort);
SetGnlAssocFormalPort (AssocI, Var);

if (!GnlCheckSamePortWidth (Var, GnlAssocActualPort(AssocI)))
{
    fprintf (stderr,
" ERROR (%d): Instance <%s> has different formal and actual widths\n",
        GnlUserComponentLineNumber (UserCompo),
        GnlUserComponentInstName (UserCompo));
    return (GNL_BAD_INSTANCE_INTERFACE);
}

if (!GnlCheckCorrectPortDir (Var, GnlAssocActualPort(AssocI)))
    return (GNL_BAD_INSTANCE_INTERFACE);

}

/* We indicate now that the formal objects are GNL_VAR and not char* */
SetGnlUserComponentFormalType (UserCompo, GNL_FORMAL_VAR);

return (GNL_OK);
}

/*
 * GnlUpdateNCheckHierarchyInterfaceRec                         */
/*
 * Verify each component connection recursively.           */
/*
GNL_STATUS GnlUpdateNCheckHierarchyInterfaceRec (Nw, Gnl, ListGnls)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    BLIST          ListGnls;
{
    BLIST          Components;
    int            j;
    GNL_COMPONENT   ComponentJ;
    GNL_USER_COMPONENT UserCompoJ;
    GNL             GnlCompo;
    GNL_STATUS      Status;
    BLIST          AuxList;
    char           CellPath[256];
    BLIST          ListCells;
    int            k;
    GNL             GnlCellK;
    FILE           *File;
}

```

gnlhier.c

```
/* This Gnl has been already processed. */  
if (GnlTag (Gnl) == GnlNetworkTag (Nw))  
    return (GNL_OK);  
  
SetGnlTag (Gnl, GnlNetworkTag (Nw));  
  
Components = GnlComponents (Gnl);  
if (!Components)  
    return (GNL_OK);  
  
for (j=0; j<BListSize (Components); j++)  
{  
    ComponentJ = (GNL_COMPONENT) BListElt (Components, j);  
    if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)  
        continue;  
  
    UserCompoJ = (GNL_USER_COMPONENT) ComponentJ;  
  
    /* Looking for the Gnl definition of 'UserCompoJ'. */  
    GnlCompo = GnlGetGnlFromUserCompoName (ListGnls,  
                                            GnlUserComponentName (UserCompoJ));  
    if (!GnlCompo)  
        continue;  
  
    SetGnlRefCount (GnlCompo, GnlRefCount (GnlCompo)+1);  
  
    SetGnlUserComponentGnlDef (UserCompoJ, GnlCompo);  
  
    if (GnlCheckNUpdateComponentInterface (UserCompoJ, GnlCompo))  
        return (GNL_BAD_INSTANCE_INTERFACE);  
  
    if ((Status = GnlUpdateNCheckHierarchyInterfaceRec (Nw, GnlCompo,  
                                                       ListGnls)))  
        return (Status);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlUpdateNCheckHierarchyInterface */  
/*-----*/  
GNL_STATUS GnlUpdateNCheckHierarchyInterface (Nw, Gnl, ListGnls)  
    GNL_NETWORK Nw;  
    GNL Gnl;  
    BLIST ListGnls;  
{  
    int i;  
    GNL GnlI;  
  
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);  
  
    /* Resetting Ref count for each Gnl. */  
    for (i=0; i<BListSize (ListGnls); i++)
```

gnlhier.c

```
{  
    GnI = (GNL)BListElt (ListGnls, i);  
    if (GnlResizeHashNames (GnI))  
        return (GNL_MEMORY_FULL);  
  
    SetGnlRefCount (GnI, 0);  
}  
  
if (GnlUpdateNCheckHierarchyInterfaceRec (Nw, Gnl, ListGnls))  
    return (GNL_MEMORY_FULL);  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlReadBlackBoxCellsRec */  
/*-----*/  
static BLIST GnlFileNotFound; /* to store names of files not found */  
  
GNL_STATUS GnlReadBlackBoxCellsRec (Nw, Gnl, ListGnls, PathDir,  
                                    Extension)  
{  
    GNL_NETWORK      Nw;  
    GNL             Gnl;  
    BLIST          ListGnls;  
    char            *PathDir;  
    char            *Extension;  
  
    BLIST          Components;  
    int             j;  
    GNL_COMPONENT   ComponentJ;  
    GNL_USER_COMPONENT UserCompoJ;  
    GNL             GnlCompo;  
    GNL_STATUS      Status;  
    BLIST          AuxList;  
    char            CellPath[256];  
    BLIST          ListCells;  
    int             k;  
    GNL             GnlCellK;  
    FILE           *File;  
    char            *NewPath;  
  
    /* This Gnl has been already processed.  
     * if (GnlTag (Gnl) == GnlNetworkTag (Nw))  
     *     return (GNL_OK);  
  
    SetGnlTag (Gnl, GnlNetworkTag (Nw));  
  
    Components = GnlComponents (Gnl);  
    if (!Components)  
        return (GNL_OK);  
  
    for (j=0; j<BListSize (Components); j++)  
    {  
        ComponentJ = (GNL_COMPONENT)BListElt (Components, j);  
    }  
}
```

```

if (GnlComponentType (ComponentJ) != GNL_USER_COMPO)
    continue;

UserCompoJ = (GNL_USER_COMPONENT) ComponentJ;

/* Looking for the Gnl definition of 'UserCompoJ'. */
GnlCompo = GnlGetGnlFromUserCompoName (ListGnls,
                                         GnlUserComponentName (UserCompoJ));

/* If we do not find the Gnl definition then we will analyze the*/
/* file having the name "PathDir/ComponentName.Extension". */
if (!GnlCompo)
{
    sprintf (CellPath, "%s/%s.%s", PathDir,
             GnlUserComponentName (UserCompoJ), Extension);

    /* If we already know that this file does not exist we */
    /* continue. */
    if (BListMemberOfList (GnlFileNotFound, CellPath,
                           StringIdentical))
        continue;

    if ((File = fopen (CellPath, "r")) == NULL)
    {
        fprintf (stderr, " WARNING: cannot find file '%s'\n",
                 CellPath);
        if (GnlStrCopy (CellPath, &NewPath))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlFileNotFound, (int)NewPath))
            return (GNL_MEMORY_FULL);
        continue;
    }
    fclose (File);

    fprintf (stderr, " Analyzing cell [%s] in file '%s'\n",
             GnlUserComponentName (UserCompoJ),
             CellPath);
    /* we expect to read small descriptions ... */
    if (GnlReadSmall (CellPath, &ListCells))
        return (GNL_MEMORY_FULL);

    /* Looking for the Gnl definition of 'UserCompoJ' in the */
    /* new list 'ListCells'. */
    GnlCompo = GnlGetGnlFromUserCompoName (ListCells,
                                         GnlUserComponentName (UserCompoJ));

    /* We append 'ListCells' to list 'ListGnls'. */
    for (k=0; k<BListSize (ListCells); k++)
    {
        GnlCellK = (GNL)BListElt (ListCells, k);
        if (BListAddElt (ListGnls, (int)GnlCellK))
            return (GNL_MEMORY_FULL);
    }
    BListQuickDelete (&ListCells);

    /* even after reading the new cells we do not find the Gnl */
    /* definition. */
}

```

gnlhier.c

```
if (!GnlCompo)
{
    continue;
}
}

SetGnlUserComponentGnlDef (UserCompoJ, GnlCompo);

if (GnlReadBlackBoxCellsRec (Nw, GnlCompo, ListGnls, PathDir,
                             Extension))
    return (Status);

}

return (GNL_OK);
}

/*-----*/
/* GnlReadBlackBoxCells                               */
/*-----*/
GNL_STATUS GnlReadBlackBoxCells (Nw, TopGnl, ListGnls, PathDir, Extension)
{
    GNL_NETWORK      Nw;
    GNL             TopGnl;
    BLIST           ListGnls;
    char            *PathDir;
    char            *Extension;
{
    char            *FileNameI;
    int             i;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    if (BListCreate (&GnlFileNotFound))
        return (GNL_MEMORY_FULL);

    if (GnlReadBlackBoxCellsRec (Nw, TopGnl, ListGnls, PathDir, Extension))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlFileNotFound); i++)
    {
        FileNameI = (char*)BListElt (GnlFileNotFound, i);
        free (FileNameI);
    }
    BListQuickDelete (&GnlFileNotFound);

    return (GNL_OK);
}

/*-----*/
/* GnlPrintTopLevels                                */
/*-----*/
/* Must call previously 'SortTopLevelModules' to print out the correct */
/* Top level modules.                                         */
/*-----*/
void GnlPrintTopLevels (ListGnls)
{
    BLIST           ListGnls;
```

```

{
    int             i;
    GNL            GnlI;

    fprintf (stderr, "\n Top Module(s): \n");
    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnlI = (GNL)BListElt (ListGnls, i);
        if (!GnlRefCount (GnlI))
            fprintf (stderr, "          o [%s]\n", GnlName (GnlI));
    }
    fprintf (stderr, "\n");
}

/*-----*/
/* GnlPrintHierarchyRec                               */
/*-----*/
void GnlPrintHierarchyRec (File, TopGnl, ListGnls, Trace, Last)
    FILE           *File;
    GNL            TopGnl;
    BLIST          ListGnls;
    char           *Trace;
    int             Last;
{
    int             i;
    int             j;
    int             k;
    int             LengthStr1;
    int             LengthStr2;
    BLIST          Components;
    GNL_COMPONENT  ComponentI;
    GNL            GnlJ;
    GNL_USER_COMPONENT UserCompoI;
    int             Last1;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;

    Components = GnlComponents (TopGnl);
    if (!Components || (BListSize (Components) == 0))
        return;

    LengthStr1 = strlen (Trace);
    Trace[LengthStr1] = '|';

    Trace[LengthStr1+1] = '\0';

    for (i=0; i<BListSize (Components); i++)
    {
        fprintf (File, Trace);
        fprintf (File, "\n");
        Trace [LengthStr1] = 'o';
        fprintf (File, Trace);
        Trace [LengthStr1] = '|';
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);

```

```

if (Last && (i == BListSize (Components)-1))
{
    k = strlen (Trace)-1;
    while (k >= 0)
    {
        if (Trace[k] == '|')
        {
            Trace[k] = ' ';
            break;
        }
        k--;
    }
}

Last1 = (i == BListSize (Components)-1);

switch (GnlComponentType (ComponentI)) {
    case GNL_USER_COMPO:
        UserCompoI = (GNL_USER_COMPONENT)ComponentI;
        fprintf (File, "-----o[%s] %s",
                 GnlUserName (UserCompoI),
                 GnlUserComponentInstName (UserCompoI));
        for (j=0; j<BListSize (ListGnls); j++)
        {
            GnlJ = (GNL)BListElt (ListGnls, j);
            if (!strcmp (GnlName (GnlJ),
                         GnlUserComponentName (UserCompoI)))
                break;
        }
        if (j != BListSize (ListGnls))
        {
            fprintf (File, "\n");
            LengthStr2 = strlen (Trace);
            for (k=0; k<6; k++)
                Trace[LengthStr2+k] = ' ';
            Trace[LengthStr2+k] = '\0';
            GnlPrintHierarchyRec (File, GnlJ, ListGnls, Trace,
                                  Last1);
            Trace[LengthStr2] = '\0';
        }
        else
        {
            fprintf (File, " (black box)\n");
        }
        break;

    case GNL_SEQUENTIAL_COMPO:
        SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
        fprintf (File, "-----o[%s] %s\n",
                 GnlSeqName (GnlSequentialCompoOp (SeqCompo)),
                 GnlSequentialCompoInstName (SeqCompo));
        break;

    case GNL_TRISTATE_COMPO:
        fprintf (File, "-----o[TRISTATE]\n");
        break;
}

```

gnlhier.c

```
case GNL_MACRO_COMPO:
    fprintf (File, "-----o[MACRO]\n");
    break;

case GNL_BUF_COMPO:
    fprintf (File, "-----o[BUF]\n");
    break;

default:
    break;
}

Trace[LengthStr1] = '\0';
}

/*-----
/* GnlPrintHierarchy
*/
void GnlPrintHierarchy (File, TopGnl, ListGnls)
FILE          *File;
GNL           TopGnl;
BLIST         ListGnls;
{
    char        Trace[2000];

    Trace[0] = ' ';
    Trace[1] = ' ';
    Trace[2] = '\0';
    fprintf (File, " [%s]\n", GnlName (TopGnl));
    GnlPrintHierarchyRec (File, TopGnl, ListGnls, Trace, 1);
}

/*-----
/* GnlCreateNetwork
*/
/* 'TopGnl' is the the top level module of the network
*/
GNL_STATUS GnlCreateNetwork (TopGnl, NewNetwork)
GNL           TopGnl;
GNL_NETWORK   *NewNetwork;
{
    BLIST     NewList;

    if (((*NewNetwork) = (GNL_NETWORK)
        calloc (1, sizeof(GNL_NETWORK_REC))) ==NULL)
        return (GNL_MEMORY_FULL);

    SetGnlNetworkTopGnl ((*NewNetwork), TopGnl);

    if (BListCreate (&NewList))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
```

gnlhier.c

```
/*-----*/
/* GnlBindVarHook */-----*/
void GnlBindVarHook (Formal, Actual)
    GNL_VAR Formal;
    GNL_VAR Actual;
{

    if (Actual == NULL)
    {
        fprintf (stderr, " ERROR: hook to bind is NULL\n");
        exit (1);
    }

    SetGnlVarHook (Formal, Actual);
}

/*-----*/
/* GnlBindAssoc */-----*/
/* Binds the Hook field of the 'GnlAssocFormalPort' of 'Assoc' with the */
/* 'GnlAssocActualPort' of Assoc. In case of buses we bind the sub-var */
/* beetwen the formal and the actual. */-----*/
/*-----*/
GNL_STATUS GnlBindAssoc (Gnl, Assoc, UserCompo)
    GNL          Gnl;
    GNL_ASSOC    Assoc;
    GNL_USER_COMPONENT UserCompo;
{
    GNL_VAR Formal;
    GNL_VAR Actual;
    BLIST ListSplitActuals;
    int LeftFormIndex;
    int RightFormIndex;
    int LeftActIndex;
    int RightActIndex;
    int i;
    GNL_VAR SplitActualI;
    GNL_STATUS GnlStatus;
    char *IndexFormalName;
    char *IndexActualName;
    GNL_VAR IndexFormalVar;
    GNL_VAR IndexActualVar;
    GNL       GnlCompo;

    Formal = GnlAssocFormalPort (Assoc);
    Actual = GnlAssocActualPort (Assoc);

    /* We hook the global formal which can be a Bus or a single bit */
    /* signal. */-----*/
    GnlBindVarHook (Formal, Actual);

    /* If var is a GNL_VAR and not a GNL_NODE (GNL_CONCAT) */
    if (GnlVarIsVar (Actual))
    {
```

```

/* If single bit signals */  

if (GnlVarRangeSize (Formal) == 1)  

{  

    GnlBindVarHook (Formal, Actual);  

    return (GNL_OK);  

}  
  

GnlCompo = GnlUserComponentGnlDef (UserCompo);  
  

LeftFormIndex = GnlVarMsb (Formal);  

RightFormIndex = GnlVarLsb (Formal);  

LeftActIndex = GnlVarMsb (Actual);  

RightActIndex = GnlVarLsb (Actual);  

if (LeftFormIndex > RightFormIndex)  

{  

    LeftFormIndex = GnlVarLsb (Formal);  

    RightFormIndex = GnlVarMsb (Formal);  

    LeftActIndex = GnlVarLsb (Actual);  

    RightActIndex = GnlVarMsb (Actual);  

}  
  

/* We bind each bus bit per bit. */  

/* we have always 'LeftFormIndex' >= 'RightFormIndex' */  

for (i=LeftFormIndex; i<=RightFormIndex; i++)  

{  

    if (GnlVarIndexName (Formal, i,  

                         &IndexFormalName))  

        return (GNL_MEMORY_FULL);  
  

    if ((GnlStatus = GnlGetVarFromName (GnlCompo,  

                                         IndexFormalName, &IndexFormalVar)))  

    {  

        if (GnlStatus == GNL_VAR_NOT_EXISTS)  

        {  

            fprintf (stderr,  

" ERROR (%d): Formal index <%s> not defined in [%s]\n",
GnlUserComponentLineNumber (UserCompo),
IndexFormalName, GnlName (GnlCompo));
free (IndexFormalName);
return (GNL_VAR_NOT_EXISTS);
}
return (GNL_MEMORY_FULL);
}  
  

    free (IndexFormalName);
    if (GnlVarIndexName (Actual, LeftActIndex,
                         &IndexActualName))
        return (GNL_MEMORY_FULL);  
  

    if ((GnlStatus = GnlGetVarFromName (Gnl,
                                         IndexActualName, &IndexActualVar)))  

    {
        if (GnlStatus == GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr,
" ERROR (%d): Actual index <%s> not defined in [%s]\n",
GnlUserComponentLineNumber (UserCompo),
IndexActualName);
            free (IndexActualName);
            return (GNL_VAR_NOT_EXISTS);
        }
    }
}
```

```

        IndexActualName, GnlName (GnlCompo));
        free (IndexActualName);
        return (GNL_VAR_NOT_EXISTS);
    }
    return (GNL_MEMORY_FULL);
}
free (IndexActualName);

GnlBindVarHook (IndexFormalVar, IndexActualVar);

if (LeftActIndex > RightActIndex)
    LeftActIndex--;
else
    LeftActIndex++;
}

return (GNL_OK);
}

GnlCompo = GnlUserComponentGnlDef (UserCompo);

/* It is then a Formal Bus and a GNL_CONCAT expression. We know that */
/* both have the same width.                                         */
ListSplitActuals = GnlNodeSons ((GNL_NODE)Actual);
LeftFormIndex = GnlVarMsb (Formal);
RightFormIndex = GnlVarLsb (Formal);
for (i=0; i<BListSize (ListSplitActuals); i++)
{
    SplitActualI = (GNL_VAR)BListElt (ListSplitActuals, i);

    /* 'IndexFormalName' is a new string in memory.                  */
    if (GnlVarIndexName (Formal, LeftFormIndex, &IndexFormalName))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlGetVarFromName (GnlCompo,
                                         IndexFormalName, &IndexFormalVar)))
    {
        if (GnlStatus == GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr,
                    " ERROR (%d): Formal index <%s> not defined in [%s]\n",
                    GnlUserComponentLineNumber (UserCompo),
                    IndexFormalName, GnlName (GnlCompo));
            free (IndexFormalName);
            return (GNL_VAR_NOT_EXISTS);
        }
        return (GNL_MEMORY_FULL);
    }

    free (IndexFormalName);

    GnlBindVarHook (IndexFormalVar, SplitActualI);

    if (LeftFormIndex > RightFormIndex)
        LeftFormIndex--;
    else

```

gnlhier.c

```
        LeftFormIndex++;
    }

    return (GNL_OK);
}

/*-----
/* GnlBindActualVarOfInterface                               */
/*-----*/
/* We bind the actual variable thru the Hook field. This field will      */
/* point on the corresponding formal var.                                */
/*-----*/
GNL_STATUS GnlBindActualVarOfInterface (Gnl, Interface, UserCompo)
{
    GNL             Gnl;
    BLIST           Interface;
    GNL_USER_COMPONENT UserCompo;
{
    int            i;
    GNL_ASSOC      AssocI;
    GNL_STATUS     Status;

    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if ((Status = GnlBindAssoc (Gnl, AssocI, UserCompo)))
            return (Status);
    }

    return (GNL_OK);
}

/*-----
/* GnlSplitNewLocalAndBind                                     */
/*-----*/
/* Both 'FormVar' and 'ActVar' are buses and local variables.          */
/* This procedure split the bus 'NewVar' into spli signals and bind      */
/* these split signals with the split of 'Var'.                         */
/*-----*/
GNL_STATUS GnlSplitNewLocalAndBind (Gnl, GnlCompo, FormVar, ActVar)
{
    GNL             Gnl;
    GNL             GnlCompo;
    GNL_VAR        FormVar;
    GNL_VAR        ActVar;
{
    int            LeftFormIndex;
    int            RightFormIndex;
    int            LeftActIndex;
    int            RightActIndex;
    int            i;
    char           *IndexFormalName;
    char           *IndexActualName;
    GNL_VAR       IndexFormalVar;
    GNL_VAR       IndexActualVar;
    GNL_STATUS    GnlStatus;
```

```

LeftFormIndex = GnlVarMsb (FormVar);
RightFormIndex = GnlVarLsb (FormVar);
LeftActIndex = GnlVarMsb (ActVar);
RightActIndex = GnlVarLsb (ActVar);

if (LeftFormIndex > RightFormIndex)
{
    LeftFormIndex = GnlVarLsb (FormVar);
    RightFormIndex = GnlVarMsb (FormVar);
    LeftActIndex = GnlVarLsb (ActVar);
    RightActIndex = GnlVarMsb (ActVar);
}

/* We bind each bus bit per bit. */ /* we have always 'LeftFormIndex' >= 'RightFormIndex' */
for (i=LeftFormIndex; i<=RightFormIndex; i++)
{
    if (GnlVarIndexName (FormVar, i, &IndexFormalName))
        return (GNL_MEMORY_FULL);

    if (GnlCreateUniqueVar (Gnl, IndexFormalName, &IndexFormalVar))
        return (GNL_MEMORY_FULL);
    SetGnlVarDir (IndexFormalVar, GnlVarDir (FormVar));

    /* This var is the result of a Bus expansion and is not original*/
    SetGnlVarType (IndexFormalVar, GNL_VAR_EXPANSED);

    if (BListAddElt (GnlLocals (Gnl), (int)IndexFormalVar))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl) + 1);

    /* Extracting the corresponding split bus actual signal */ /* */
    if (GnlVarIndexName (ActVar, LeftActIndex, &IndexActualName))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlGetVarFromName (GnlCompo, IndexActualName,
                                         &IndexActualVar)))
    {
        if (GnlStatus == GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr,
                     " ERROR (%d): Actual index <%s> not defined in [%s]\n",
                     GnlName (GnlCompo),
                     IndexActualName, GnlName (GnlCompo));
            free (IndexActualName);
            return (GNL_VAR_NOT_EXISTS);
        }
        return (GNL_MEMORY_FULL);
    }
    free (IndexActualName);

    GnlBindVarHook (IndexActualVar, IndexFormalVar);

    if (LeftActIndex > RightActIndex)

```

```

        LeftActIndex--;
    else
        LeftActIndex++;
    }

    return (GNL_OK);
}

/*-----
/* GnlBindAndCreateLocalVar
/*-----
/* This procedure takes each local variable of 'GnlCompo' and creates */
/* equivalent variables in 'Gnl'. Then it binds these new variable of */
/* 'Gnl' with the ones of 'GnlCompo'.
/*-----*/
GNL_STATUS GnlBindAndCreateLocalVar (Gnl, GnlCompo)
    GNL          Gnl;
    GNL          GnlCompo;
{
    BLIST      HashTableNames;
    int         i;
    int         j;
    BLIST      BucketI;
    GNL_VAR    VarJ;
    GNL_VAR    NewVar;
    char        *NewLoc;

    /* we reset the hook field of all the Vars in the hash table name of */
    /* 'GnlCompo'. */
    HashTableNames = GnlHashNames (GnlCompo);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST)BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);
            SetGnlVarHook (VarJ, NULL);
        }
    }

    HashTableNames = GnlHashNames (GnlCompo);

    /* First of all we bind the local buses and then we bind the single */
    /* variables. For instance if B is a bus [2:0] then we build a new */
    /* local variable L of same size and then split both into bits and */
    /* bind each bits: b[2] -> L[2], B[1] -> L[1], ... */
    /* If we do not do that we may loose the bus relation between the */
    /* signals: b[2] -> x, b[1] -> y, ... */
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST)BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);

```

```

/* If 'VarJ' is a bus. */  

/* If 'VarJ' has a hook then this is because it is a split */  

/* bus variable which has been previously binded and then */  

/* the job is done. */  

if ((GnlVarHook (VarJ) == NULL) &&  

    (GnlVarRangeSize (VarJ) > 1) &&  

    (GnlVarDir (VarJ) != GNL_VAR_INPUT) &&  

    (GnlVarDir (VarJ) != GNL_VAR_OUTPUT) &&  

    (GnlVarDir (VarJ) != GNL_VAR_INOUT))  

{  

    /* It is a local var. */  

    if (GnlCreateUniqueVar (Gnl, GnlVarName (VarJ),  

                           &NewVar))  

        return (GNL_MEMORY_FULL);  

    if (BListAddElt (GnlLocals (Gnl), (int) NewVar))  

        return (GNL_MEMORY_FULL);  

    SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);  

    SetGnlVarDir (NewVar, GnlVarDir (VarJ));  

    SetGnlVarType (NewVar, GnlVarType (VarJ));  

    SetGnlVarMsb (NewVar, GnlVarMsb (VarJ));  

    SetGnlVarLsb (NewVar, GnlVarLsb (VarJ));  

    if (GnlVarLocation (VarJ))  

    {  

        if (GnlStrCopy (GnlVarLocation (VarJ), &NewLoc))  

            return (GNL_MEMORY_FULL);  

        SetGnlVarLocation (NewVar, NewLoc);  

    }  

    /* We bind the hook field of this newvar to the 'VarJ' */  

    GnlBindVarHook (VarJ, NewVar);  

    if (GnlSplitNewLocalAndBind (Gnl, GnlCompo, NewVar,  

                               VarJ))  

        return (GNL_MEMORY_FULL);  

}
}  

}  

for (i=0; i<BListSize (HashTableNames); i++)  

{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
  

        /* If the local var has no Hook yet then it is not a bus */  

        /* variable nor a split bus variable but a single bit var. */  

        if ((GnlVarHook (VarJ) == NULL) &&  

            (GnlVarDir (VarJ) != GNL_VAR_INPUT) &&  

            (GnlVarDir (VarJ) != GNL_VAR_OUTPUT) &&  

            (GnlVarDir (VarJ) != GNL_VAR_INOUT))
    {
        /* It is a local var. */  

        if (GnlCreateUniqueVar (Gnl, GnlVarName (VarJ),

```

```

        &NewVar))
    return (GNL_MEMORY_FULL);
if (BListAddElt (GnlLocals (Gnl), (int) NewVar))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);

SetGnlVarDir (NewVar, GnlVarDir (VarJ));
SetGnlVarType (NewVar, GnlVarType (VarJ));
SetGnlVarMsb (NewVar, GnlVarMsb (VarJ));
SetGnlVarLsb (NewVar, GnlVarLsb (VarJ));

if (GnlVarLocation (VarJ))
{
    if (GnlStrCopy (GnlVarLocation (VarJ), &NewLoc))
        return (GNL_MEMORY_FULL);
    SetGnlVarLocation (NewVar, NewLoc);
}

/* We bind the hook field of this newvar to the 'VarJ'      */
GnlBindVarHook (VarJ, NewVar);
}
}

return (GNL_OK);
}

/*
/* GnlDuplicateNodeForNewGnl                               */
/*
GNL_STATUS GnlDuplicateNodeForNewGnl (NewGnl, GnlNode, NewNode)
    GNL           NewGnl;
    GNL_NODE     GnlNode;
    GNL_NODE *NewNode;
{
    int             i;
    GNL_VAR       Var;
    GNL_VAR     ActualVar;
    BLIST       Sons;
    GNL_NODE   SonI;
    BLIST       NewList;
    GNL_NODE NewSonI;

switch (GnlNodeOp (GnlNode)) {

case GNL_CONSTANTE:
    if (GnlCreateNode (NewGnl, GNL_CONSTANTE, NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons ((*NewNode), GnlNodeSons (GnlNode));
    SetGnlNodeLineNumber ((*NewNode),
                          GnlNodeLineNumber (GnlNode));
    return (GNL_OK);

case GNL_VARIABLE:
    Var = (GNL_VAR) GnlNodeSons (GnlNode);
    if (GnlVarIsVss (Var))

```

```

{
    if (GnlCreateNodeVss (NewGnl, NewNode))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
if (GnlVarIsVdd (Var))
{
    if (GnlCreateNodeVdd (NewGnl, NewNode))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
/* we pick up the corresponding Actual var of 'Var'      */
ActualVar = (GNL_VAR) GnlVarHook (Var);
if (!ActualVar)
{
    /* If already replaced in the GNL_NODE tree           */
    if (GnlGetVarFromName (NewGnl, GnlVarName (Var),
                           &ActualVar))
    {
        fprintf (stderr,
"ERROR: <%s> has no associated var during flattening\n",
        GnlVarName (Var));
        exit (1);
    }
}
if (GnlCreateNodeForVar (NewGnl, ActualVar, NewNode))
    return (GNL_MEMORY_FULL);

return (GNL_OK);

default:
    Sons = GnlNodeSons (GnlNode);
    if (GnlCreateNode (NewGnl, GnlNodeOp (GnlNode), NewNode))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (BListSize (Sons), &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons ((*NewNode), NewList);

    for (i=0; i<BListSize (Sons); i++)
    {
        SonI = (GNL_NODE) BListElt (Sons, i);
        if (GnlDuplicateNodeForNewGnl (NewGnl, SonI, &NewSonI))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlNodeSons ((*NewNode)),
                         (int) NewSonI))
            return (GNL_MEMORY_FULL);
    }
    return (GNL_OK);
}

/*
 * GnlSubstituteNodeForNewGnl
 */
GNL_STATUS GnlSubstituteNodeForNewGnl (NewGnl, GnlNode, NewNode)
    GNL             NewGnl;
    GNL_NODE        GnlNode;

```

gnlhier.c

```

    GNL_NODE *NewNode;
}

int           i;
GNL_VAR Var;
GNL_VAR ActualVar;
BLIST Sons;
GNL_NODE SonI;
BLIST NewList;
GNL_NODE NewSonI;
GNL_VAR NewVar;

switch (GnlNodeOp (GnlNode)) {

    case GNL_CONSTANTE:
        *NewNode = GnlNode;
        return (GNL_OK);

    case GNL_VARIABLE:
        Var = (GNL_VAR)GnlNodeSons (GnlNode);
        if (GnlVarIsVss (Var))
        {
            *NewNode = GnlNode;
            return (GNL_OK);
        }
        if (GnlVarIsVdd (Var))
        {
            *NewNode = GnlNode;
            return (GNL_OK);
        }

        /* we pick up the corresponding new var of 'Var' */
        NewVar = (GNL_VAR)GnlVarHook (Var);
        if (!NewVar)
        {
            /* If already replaced in the GNL_NODE tree */ *
            if (GnlGetVarFromName (NewGnl, GnlVarName (Var), &NewVar))
            {
                fprintf (stderr,
                "ERROR: <%s> has no associated var during flattening\n",
                GnlVarName (Var));
                exit (1);
            }
        }
        SetGnlNodeSons (GnlNode, (BLIST)NewVar);
        *NewNode = GnlNode;

        return (GNL_OK);

    default:
        Sons = GnlNodeSons (GnlNode);
        for (i=0; i<BListSize (Sons); i++)
        {
            SonI = (GNL_NODE)BListElt (Sons, i);
            if (GnlSubstituteNodeForNewGnl (NewGnl, SonI, &NewSonI))
                return (GNL_MEMORY_FULL);
            BListElt (Sons, i) = (int)NewSonI;
        }
}

```

```

        }
        *NewNode = GnlNode;
        return (GNL_OK);
    }
}

/*-----*/
/* GnlDuplicateAssoc */
/*-----*/
GNL_STATUS GnlDuplicateAssoc (Gnl, Assoc, NewAssoc)
{
    GNL          Gnl;
    GNL_ASSOC    Assoc;
    GNL_ASSOC    *NewAssoc;
{
    int          i;
    GNL_VAR     Formal;
    GNL_VAR     Actual;
    GNL_VAR     SplitActualI;
    GNL_VAR     SubstituteActualI;
    BLIST       ListSplitActuals;
    BLIST       NewList;
    GNL_NODE   NewNode;

    Formal = GnlAssocFormalPort (Assoc);
    Actual = GnlAssocActualPort (Assoc);

    /* If it is an open port */
    if (!Actual)
    {
        if (GnlCreateAssoc (NewAssoc))
            return (GNL_MEMORY_FULL);
        SetGnlAssocFormalPort ((*NewAssoc), Formal);

        return (GNL_OK);
    }

    /* both Formal and Actual are 1-bit width signals */
    if (GnlVarIsVar (Actual))
    {
        if (GnlCreateAssoc (NewAssoc))
            return (GNL_MEMORY_FULL);
        SetGnlAssocFormalPort ((*NewAssoc), Formal);

        /* we substitute the actual signal in the component instantiation*/
        if (GnlVarHook (Actual) == NULL)
        {
            fprintf (stderr, " ERROR: signal has no hook\n");
            exit (1);
        }

        SetGnlAssocActualPort ((*NewAssoc),
                               (GNL_VAR) GnlVarHook (Actual));
        return (GNL_OK);
    }

    if (GnlCreateAssoc (NewAssoc))
}

```

gnlhier.c

```
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort ((*NewAssoc), Formal);

/* It is then a Formal Bus and a GNL_CONCAT expression. We know that */
/* both have the same width.                                         */
ListSplitActuals = GnlNodeSons ((GNL_NODE)Actual);

/* creating the CONCAT node                                         */
if (GnlCreateConcatNode (&NewNode))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (BListSize (ListSplitActuals), &NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (NewNode, NewList);

for (i=0; i<BListSize (ListSplitActuals); i++)
{
    SplitActualI = (GNL_VAR)BListElt (ListSplitActuals, i);

    SubstituteActualI = (GNL_VAR)GnlVarHook (SplitActualI);
    if (BListAddElt (NewList, (int)SubstituteActualI))
        return (GNL_MEMORY_FULL);
}

SetGnlAssocActualPort ((*NewAssoc), (GNL_VAR)NewNode);

return (GNL_OK);
}

/*
 * GnlDuplicateUserComponent
 */
GNL_STATUS GnlDuplicateUserComponent (Gnl, FatherCompo, UserCompo,
                                      NewCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        FatherCompo;
    GNL_USER_COMPONENT        UserCompo;
    GNL_COMPONENT             *NewCompo;

    GNL_USER_COMPONENT        NewUserCompo;
    BLIST                     NewInterface;
    BLIST                     NewList;
    BLIST                     Interface;
    int                       i;
    GNL_ASSOC                 AssocI;
    GNL_ASSOC                 NewAssoc;
    char                      *AuxName;
    char                      *NewName;

    Interface = GnlUserComponentInterface (UserCompo);
    if (BListCreateWithSize (BListSize (Interface), &NewList))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if (GnlDuplicateAssoc (Gnl, AssocI, &NewAssoc))
```

gnlhier.c

```
        return (GNL_MEMORY_FULL);
    if (BListAddElt (NewList, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
    }

    if (GnlCreateUserComponent (GnlUserComponentName (UserCompo),
                               GnlUserComponentInstName (UserCompo),
                               NULL,
                               NewList, &NewUserCompo))
        return (GNL_MEMORY_FULL);

    if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
    {
        if (!GnlUserComponentInstName (UserCompo))
        {
            if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                           &NewName))
                return (GNL_MEMORY_FULL);
        }
        else
        {
            if (GnlStrAppendStrCopy (
                    GnlUserComponentInstName (FatherCompo),
                    GnlEnvFlattenStr (), &AuxName))
                return (GNL_MEMORY_FULL);
            if (GnlStrAppendStrCopy (AuxName,
                                   GnlUserComponentInstName (UserCompo),
                                   &NewName))
                return (GNL_MEMORY_FULL);
            free (AuxName);
        }
        SetGnlUserComponentInstName (NewUserCompo, NewName);
    }

    SetGnlUserComponentFormalType (NewUserCompo,
                                  GnlUserComponentFormalType (UserCompo));
    *NewCompo = (GNL_COMPONENT)NewUserCompo;

    return (GNL_OK);
}

/*-----
/* GnlDuplicateSeqComponent
/*-----*/
GNL_STATUS GnlDuplicateSeqComponent (Gnl, FatherCompo, SeqCompo,
                                    NewCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        FatherCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT             *NewCompo;
{
    GNL_SEQUENTIAL_COMPONENT NewSeqCompo;
    char                     *AuxName;
    char                     *NewName;
```

```

if (GnlCreateSequentialComponent (GnlSequentialCompoType (SeqCompo) ,
                                &NewSeqCompo))
    return (GNL_MEMORY_FULL);

SetGnlSequentialCompoOp (NewSeqCompo,
                        GnlSequentialCompoOp (SeqCompo));

/* Replacing each signal by its hook signal. */
if (!GnlVarIsVss (GnlSequentialCompoInput (SeqCompo)) &&
    !GnlVarIsVdd (GnlSequentialCompoInput (SeqCompo)))
    SetGnlSequentialCompoInput (NewSeqCompo, (GNL_VAR) GnlVarHook (
                                GnlSequentialCompoInput (SeqCompo)));
else
    SetGnlSequentialCompoInput (NewSeqCompo,
                                GnlSequentialCompoInput (SeqCompo));

if (!GnlVarIsVss (GnlSequentialCompoOutput (SeqCompo)) &&
    !GnlVarIsVdd (GnlSequentialCompoOutput (SeqCompo)))
    SetGnlSequentialCompoOutput (NewSeqCompo, (GNL_VAR) GnlVarHook (
                                GnlSequentialCompoOutput (SeqCompo)));
else
    SetGnlSequentialCompoOutput (NewSeqCompo,
                                GnlSequentialCompoOutput (SeqCompo));

if (!GnlVarIsVss (GnlSequentialCompoClock (SeqCompo)) &&
    !GnlVarIsVdd (GnlSequentialCompoClock (SeqCompo)))
    SetGnlSequentialCompoClock (NewSeqCompo, (GNL_VAR) GnlVarHook (
                                GnlSequentialCompoClock (SeqCompo)));
else
    SetGnlSequentialCompoClock (NewSeqCompo,
                                GnlSequentialCompoClock (SeqCompo));

SetGnlSequentialCompoClockPol (NewSeqCompo,
                               GnlSequentialCompoClockPol (SeqCompo));

if (GnlSequentialCompoReset (SeqCompo) &&
    !GnlVarIsVss (GnlSequentialCompoReset (SeqCompo)) &&
    !GnlVarIsVdd (GnlSequentialCompoReset (SeqCompo)))
    SetGnlSequentialCompoReset (NewSeqCompo, (GNL_VAR) GnlVarHook (
                                GnlSequentialCompoReset (SeqCompo)));
else
    SetGnlSequentialCompoReset (NewSeqCompo,
                                GnlSequentialCompoReset (SeqCompo));

if (GnlSequentialCompoSet (SeqCompo) &&
    !GnlVarIsVss (GnlSequentialCompoSet (SeqCompo)) &&
    !GnlVarIsVdd (GnlSequentialCompoSet (SeqCompo)))
    SetGnlSequentialCompoSet (NewSeqCompo, (GNL_VAR) GnlVarHook (
                                GnlSequentialCompoSet (SeqCompo)));
else
    SetGnlSequentialCompoSet (NewSeqCompo,
                              GnlSequentialCompoSet (SeqCompo));

SetGnlSequentialCompoSetPol (NewSeqCompo,
                            GnlSequentialCompoSetPol (SeqCompo));

```

```

SetGnlSequentialCompoResetPol (NewSeqCompo,
                               GnlSequentialCompoResetPol (SeqCompo)) ;

if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
{
    if (!GnlSequentialCompoInstName (SeqCompo))
    {
        if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                        &NewName))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlStrAppendStrCopy (
                GnlUserComponentInstName (FatherCompo),
                GnlEnvFlattenStr (), &AuxName))
            return (GNL_MEMORY_FULL);
        if (GnlStrAppendStrCopy (AuxName,
                                GnlSequentialCompoInstName (SeqCompo),
                                &NewName))
            return (GNL_MEMORY_FULL);
        free (AuxName);
    }
    SetGnlSequentialCompoInstName (NewSeqCompo, NewName);
}

*NewCompo = (GNL_COMPONENT) NewSeqCompo;

return (GNL_OK);
}

/*
 * GnlDuplicateBufComponent
 */
GNL_STATUS GnlDuplicateBufComponent (Gnl, FatherCompo, BufCompo,
                                    NewCompo)
{
    GNL;
    GNL_USER_COMPONENT;
    GNL_BUF_COMPONENT;
    GNL_COMPONENT;
    GNL_BUF_COMPONENT;
    char;
    char;

    if (GnlCreateBufComponent (&NewBufCompo))
        return (GNL_MEMORY_FULL);

    /* Replacing each signal by its hook signal. */
    if (!GnlVarIsVss (GnlBufInput (BufCompo)) &&
        !GnlVarIsVdd (GnlBufInput (BufCompo)))
        SetGnlBufInput (NewBufCompo, (GNL_VAR) GnlVarHook (
            GnlBufInput (BufCompo)));
}

```

gnlhier.c

```
else
    SetGnlBufInput (NewBufCompo, GnlBufInput (BufCompo)) ;

if (!GnlVarIsVss (GnlBufOutput (BufCompo)) &&
    !GnlVarIsVdd (GnlBufOutput (BufCompo)))
    SetGnlBufOutput (NewBufCompo, (GNL_VAR) GnlVarHook (
        GnlBufOutput (BufCompo)));
else
    SetGnlBufOutput (NewBufCompo, GnlBufOutput (BufCompo));

if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
{
    if (!GnlBufInstName (BufCompo))
    {
        if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                        &NewName))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlStrAppendStrCopy (
                GnlUserComponentInstName (FatherCompo),
                GnlEnvFlattenStr (), &AuxName))
            return (GNL_MEMORY_FULL);
        if (GnlStrAppendStrCopy (AuxName,
                                GnlBufInstName (BufCompo),
                                &NewName))
            return (GNL_MEMORY_FULL);
        free (AuxName);
    }
    SetGnlTriStateInstName (NewBufCompo, NewName);
}

*NewCompo = (GNL_COMPONENT) NewBufCompo;

return (GNL_OK);
}

/*
 * GnlDuplicateTristateComponent
 */
GNL_STATUS GnlDuplicateTristateComponent (Gnl, FatherCompo, TriCompo,
                                         NewCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT         FatherCompo;
    GNL_TRISTATE_COMPONENT    TriCompo;
    GNL_COMPONENT              *NewCompo;
{
    GNL_TRISTATE_COMPONENT    NewTriCompo;
    char                      *AuxName;
    char                      *NewName;

    if (GnlCreateTristateComponent (&NewTriCompo))
        return (GNL_MEMORY_FULL);
```

```

/* Replacing each signal by its hook signal. */  

if (!GnlVarIsVss (GnlTriStateInput (TriCompo)) &&  

    !GnlVarIsVdd (GnlTriStateInput (TriCompo)))  

    SetGnlTriStateInput (NewTriCompo, (GNL_VAR) GnlVarHook (  

                                GnlTriStateInput (TriCompo)));  

else  

    SetGnlTriStateInput (NewTriCompo, GnlTriStateInput (TriCompo));  

if (!GnlVarIsVss (GnlTriStateOutput (TriCompo)) &&  

    !GnlVarIsVdd (GnlTriStateOutput (TriCompo)))  

    SetGnlTriStateOutput (NewTriCompo, (GNL_VAR) GnlVarHook (  

                                GnlTriStateOutput (TriCompo)));  

else  

    SetGnlTriStateOutput (NewTriCompo, GnlTriStateOutput (TriCompo));  

if (!GnlVarIsVss (GnlTriStateSelect (TriCompo)) &&  

    !GnlVarIsVdd (GnlTriStateSelect (TriCompo)))  

    SetGnlTriStateSelect (NewTriCompo, (GNL_VAR) GnlVarHook (  

                                GnlTriStateSelect (TriCompo)));  

else  

    SetGnlTriStateSelect (NewTriCompo, GnlTriStateSelect (TriCompo));  

SetGnlTriStateSelectPol (NewTriCompo, GnlTriStateSelectPol (TriCompo));  

SetGnlTriStateInputPol (NewTriCompo, GnlTriStateInputPol (TriCompo));  

if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)  

{
    if (!GnlTriStateInstName (TriCompo))
    {
        if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                        &NewName))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlStrAppendStrCopy (
                GnlUserComponentInstName (FatherCompo),
                GnlEnvFlattenStr (), &AuxName))
            return (GNL_MEMORY_FULL);
        if (GnlStrAppendStrCopy (AuxName,
                               GnlTriStateInstName (TriCompo),
                               &NewName))
            return (GNL_MEMORY_FULL);
        free (AuxName);
    }
    SetGnlTriStateInstName (NewTriCompo, NewName);
}
  

*NewCompo = (GNL_COMPONENT) NewTriCompo;  

return (GNL_OK);
}  

/*-----*/  

/* GnlDuplicateComponentForNewGnl */  

/*-----*/

```

gnlhier.c

```
GNL_STATUS GnlDuplicateComponentForNewGnl (Gnl, FatherCompo, Component,
                                         NewCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        FatherCompo;
    GNL_COMPONENT             Component;
    GNL_COMPONENT             *NewCompo;
}

switch (GnlComponentType (Component)) {
    case GNL_SEQUENTIAL_COMPO:
        if (GnlDuplicateSeqComponent (Gnl, FatherCompo,
                                      (GNL_SEQUENTIAL_COMPONENT) Component,
                                      NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_TRISTATE_COMPO:
        if (GnlDuplicateTristateComponent (Gnl, FatherCompo,
                                           (GNL_TRISTATE_COMPONENT) Component,
                                           NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_BUF_COMPO:
        if (GnlDuplicateBufComponent (Gnl, FatherCompo,
                                      (GNL_BUF_COMPONENT) Component,
                                      NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_MACRO_COMPO:
        return (GNL_OK);

    case GNL_USER_COMPO:
        if (GnlDuplicateUserComponent (Gnl, FatherCompo,
                                      (GNL_USER_COMPONENT) Component,
                                      NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    default:
        GnlError (20 /* component unknown */);
        return (GNL_MEMORY_FULL);      /* to change */
    }
}

/*-----*/
/* GnlSubstituteAssoc                                */
/*-----*/
GNL_STATUS GnlSubstituteAssoc (Gnl, Assoc, NewAssoc)
{
    GNL                      Gnl;
    GNL_ASSOC                 Assoc;
    GNL_ASSOC                 *NewAssoc;
{
    int                      i;
```

gnlhier.c

```
GNL_VAR Formal;
GNL_VAR Actual;
GNL_VAR SplitActualI;
GNL_VAR SubstituteActualI;
BLIST ListSplitActuals;

Formal = GnlAssocFormalPort (Assoc);
Actual = GnlAssocActualPort (Assoc);

if (!Actual)
{
    return (GNL_OK);
}

/* both Formal and Actual are 1-bit width signals */
if (GnlVarIsVar (Actual))
{
    /* we substitute the actual signal in the component instantiation*/
    if (GnlVarHook (Actual) == NULL)
    {
        fprintf (stderr, " ERROR: signal has no hook\n");
        exit (1);
    }

    SetGnlAssocActualPort (Assoc, (GNL_VAR) GnlVarHook (Actual));
    return (GNL_OK);
}

/* It is then a Formal Bus and a GNL_CONCAT expression. We know that */
/* both have the same width. */
ListSplitActuals = GnlNodeSons ((GNL_NODE) Actual);

for (i=0; i<BListSize (ListSplitActuals); i++)
{
    SplitActualI = (GNL_VAR) BListElt (ListSplitActuals, i);

    SubstituteActualI = (GNL_VAR) GnlVarHook (SplitActualI);
    BListElt (ListSplitActuals, i) = (int) SubstituteActualI;
}

return (GNL_OK);
}

/*
 * GnlSubstituteUserComponent
 */
GNL_STATUS GnlSubstituteUserComponent (Gnl, FatherCompo, UserCompo,
                                         NewCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT      FatherCompo;
    GNL_USER_COMPONENT      UserCompo;
    GNL_COMPONENT           *NewCompo;
    {
        BLIST          Interface;
        int            i;
        GNL_ASSOC      AssocI;
```

gnlhier.c

```
char *AuxName;
char *NewName;

Interface = GnlUserComponentInterface (UserCompo);

for (i=0; i<BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListEl (Interface, i);
    if (GnlSubstituteAssoc (Gnl, AssocI))
        return (GNL_MEMORY_FULL);
}

if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
{
    if (!GnlUserComponentInstName (UserCompo))
    {
        if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                        &NewName))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlStrAppendStrCopy (
                GnlUserComponentInstName (FatherCompo),
                GnlEnvFlattenStr (), &AuxName))
            return (GNL_MEMORY_FULL);
        if (GnlStrAppendStrCopy (AuxName,
                                GnlUserComponentInstName (UserCompo),
                                &NewName))
            return (GNL_MEMORY_FULL);
        free (AuxName);
        free (GnlUserComponentInstName (UserCompo));
    }
}

SetGnlUserComponentInstName (UserCompo, NewName);
}

*NewCompo = (GNL_COMPONENT)UserCompo;

return (GNL_OK);
}

/*-----*/
/* GnlSubstituteSeqComponent */
/*-----*/
GNL_STATUS GnlSubstituteSeqComponent (Gnl, FatherCompo, SeqCompo,
                                      NewCompo)
{
    GNL                               Gnl;
    GNL_USER_COMPONENT                 FatherCompo;
    GNL_SEQUENTIAL_COMPONENT          SeqCompo;
    GNL_COMPONENT                      *NewCompo;
{
    char *AuxName;
    char *NewName;
```

```

/* Replacing each signal by its hook signal. */  

if (!GnlVarIsVss (GnlSequentialCompoInput (SeqCompo)) &&  

    !GnlVarIsVdd (GnlSequentialCompoInput (SeqCompo)))  

    SetGnlSequentialCompoInput (SeqCompo, (GNL_VAR) GnlVarHook (   

                                         GnlSequentialCompoInput (SeqCompo)));  

if (!GnlVarIsVss (GnlSequentialCompoOutput (SeqCompo)) &&  

    !GnlVarIsVdd (GnlSequentialCompoOutput (SeqCompo)))  

    SetGnlSequentialCompoOutput (SeqCompo, (GNL_VAR) GnlVarHook (   

                                         GnlSequentialCompoOutput (SeqCompo)));  

if (!GnlVarIsVss (GnlSequentialCompoClock (SeqCompo)) &&  

    !GnlVarIsVdd (GnlSequentialCompoClock (SeqCompo)))  

    SetGnlSequentialCompoClock (SeqCompo, (GNL_VAR) GnlVarHook (   

                                         GnlSequentialCompoClock (SeqCompo)));  

if (GnlSequentialCompoReset (SeqCompo) &&  

    !GnlVarIsVss (GnlSequentialCompoReset (SeqCompo)) &&  

    !GnlVarIsVdd (GnlSequentialCompoReset (SeqCompo)))  

    SetGnlSequentialCompoReset (SeqCompo, (GNL_VAR) GnlVarHook (   

                                         GnlSequentialCompoReset (SeqCompo)));  

if (GnlSequentialCompoSet (SeqCompo) &&  

    !GnlVarIsVss (GnlSequentialCompoSet (SeqCompo)) &&  

    !GnlVarIsVdd (GnlSequentialCompoSet (SeqCompo)))  

    SetGnlSequentialCompoSet (SeqCompo, (GNL_VAR) GnlVarHook (   

                                         GnlSequentialCompoSet (SeqCompo)));  

if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)  

{  

    if (!GnlSequentialCompoInstName (SeqCompo))  

    {  

        if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),  

                        &NewName))  

            return (GNL_MEMORY_FULL);  

    }  

    else  

    {  

        if (GnlStrAppendStrCopy (   

                                GnlUserComponentInstName (FatherCompo),  

                                GnlEnvFlattenStr (), &AuxName))  

            return (GNL_MEMORY_FULL);  

        if (GnlStrAppendStrCopy (AuxName,  

                                GnlSequentialCompoInstName (SeqCompo),  

                                &NewName))  

            return (GNL_MEMORY_FULL);  

        free (AuxName);  

        free (GnlSequentialCompoInstName (SeqCompo));  

    }  

    SetGnlSequentialCompoInstName (SeqCompo, NewName);  

}  

*NewCompo = (GNL_COMPONENT) SeqCompo;  

    return (GNL_OK);
}  

/*-----*/  

/* GnlSubstituteBufComponent */  

/*-----*/

```

gnlhier.c

```
GNL_STATUS GnlSubstituteBufComponent (Gnl, FatherCompo, BufCompo,
                                      NewCompo)
{
    GNL                               Gnl;
    GNL_USER_COMPONENT                FatherCompo;
    GNL_BUF_COMPONENT                 BufCompo;
    GNL_COMPONENT                     *NewCompo;
{
    char                         *AuxName;
    char                         *NewName;

    /* Replacing each signal by its hook signal. */
    if (!GnlVarIsVss (GnlBufInput (BufCompo)) &&
        !GnlVarIsVdd (GnlBufInput (BufCompo)))
        SetGnlBufInput (BufCompo, (GNL_VAR) GnlVarHook (
                                         GnlBufInput (BufCompo)));
    if (!GnlVarIsVss (GnlBufOutput (BufCompo)) &&
        !GnlVarIsVdd (GnlBufOutput (BufCompo)))
        SetGnlTriStateOutput (BufCompo, (GNL_VAR) GnlVarHook (
                                         GnlBufOutput (BufCompo)));

    if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
    {
        if (!GnlBufInstName (BufCompo))
        {
            if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                            &NewName))
                return (GNL_MEMORY_FULL);
        }
        else
        {
            if (GnlStrAppendStrCopy (
                    GnlUserComponentInstName (FatherCompo),
                    GnlEnvFlattenStr (), &AuxName))
                return (GNL_MEMORY_FULL);
            if (GnlStrAppendStrCopy (AuxName,
                                    GnlBufInstName (BufCompo),
                                    &NewName))
                return (GNL_MEMORY_FULL);
            free (AuxName);
            free (GnlBufInstName (BufCompo));
        }
        SetGnlBufInstName (BufCompo, NewName);
    }
    *NewCompo = (GNL_COMPONENT) BufCompo;

    return (GNL_OK);
}

/*-----
/* GnlSubstituteTristateComponent
/*-----*/
GNL_STATUS GnlSubstituteTristateComponent (Gnl, FatherCompo, TriCompo,
                                            NewCompo)
{
    GNL                               Gnl;
```

```

GNL_USER_COMPONENT           FatherCompo;
GNL_TRISTATE_COMPONENT      TriCompo;
GNL_COMPONENT                *NewCompo;
{
    char                      *AuxName;
    char                      *NewName;

    /* Replacing each signal by its hook signal. */
    if (!GnlVarIsVss (GnlTriStateInput (TriCompo)) &&
        !GnlVarIsVdd (GnlTriStateInput (TriCompo)))
        SetGnlTriStateInput (TriCompo, (GNL_VAR) GnlVarHook (
            GnlTriStateInput (TriCompo)));
    if (!GnlVarIsVss (GnlTriStateOutput (TriCompo)) &&
        !GnlVarIsVdd (GnlTriStateOutput (TriCompo)))
        SetGnlTriStateOutput (TriCompo, (GNL_VAR) GnlVarHook (
            GnlTriStateOutput (TriCompo)));
    if (!GnlVarIsVss (GnlTriStateSelect (TriCompo)) &&
        !GnlVarIsVdd (GnlTriStateSelect (TriCompo)))
        SetGnlTriStateSelect (TriCompo, (GNL_VAR) GnlVarHook (
            GnlTriStateSelect (TriCompo)));

    if (GnlEnvFlattenInstanceName () == GNL_FLAT_HIERARCHICAL_NAME)
    {
        if (!GnlTriStateInstName (TriCompo))
        {
            if (GnlStrCopy (GnlUserComponentInstName (FatherCompo),
                            &NewName))
                return (GNL_MEMORY_FULL);
        }
        else
        {
            if (GnlStrAppendStrCopy (
                    GnlUserComponentInstName (FatherCompo),
                    GnlEnvFlattenStr (), &AuxName))
                return (GNL_MEMORY_FULL);
            if (GnlStrAppendStrCopy (AuxName,
                                    GnlTriStateInstName (TriCompo),
                                    &NewName))
                return (GNL_MEMORY_FULL);
            free (AuxName);
            free (GnlTriStateInstName (TriCompo));
        }
        SetGnlTriStateInstName (TriCompo, NewName);
    }
    *NewCompo = (GNL_COMPONENT) TriCompo;

    return (GNL_OK);
}

/*-----
/* GnlSubstituteComponentForNewGnl
/*-----*/
GNL_STATUS GnlSubstituteComponentForNewGnl (Gnl, FatherCompo, Component,
                                            NewCompo)

```

gnlhier.c

```
GNL                      Gnl;
GNL_USER_COMPONENT    FatherCompo;
GNL_COMPONENT   Component;
GNL_COMPONENT  *NewCompo;
{

switch (GnlComponentType (Component)) {
    case GNL_SEQUENTIAL_COMPO:
        if (GnlSubstituteSeqComponent (Gnl, FatherCompo,
                                       (GNL_SEQUENTIAL_COMPONENT)Component, NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_TRISTATE_COMPO:
        if (GnlSubstituteTristateComponent (Gnl, FatherCompo,
                                             (GNL_TRISTATE_COMPONENT)Component, NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_BUF_COMPO:
        if (GnlSubstituteBufComponent (Gnl, FatherCompo,
                                       (GNL_BUF_COMPONENT)Component, NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_MACRO_COMPO:
        return (GNL_OK);

    case GNL_USER_COMPO:
        if (GnlSubstituteUserComponent (Gnl, FatherCompo,
                                       (GNL_USER_COMPONENT)Component,
                                       NewCompo))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    default:
        GnlError (20 /* component unknown */);
        return (GNL_MEMORY_FULL); /* to change */
    }
}

/*
 *-----*/
/* GnlReplaceSubstituteComponentInGnl */ */
/*-----*/
/* This procedure substitutes variables of gnl corresponding to */
/* 'UserCompo' and adds this new gnl in the gnl 'Gnl'. */
/* This procedure is invoked when 'UserCompo' is the last instance to */
/* be processed. */
/* This procedure makes a side effect on the list 'ListGnls' by removing*/
/* the element 'Gnl' present in 'ListGnls'. */
/*-----*/
GNL_STATUS GnlReplaceSubstituteComponentInGnl (Gnl, UserCompo, ListGnls)
    GNL          Gnl;
    GNL_USER_COMPONENT UserCompo;
    BLIST        ListGnls;
```

```

{
    BLIST           Interface;
    GNL_STATUS     Status;
    GNL            GnlCompo;
    int             i;
    GNL_VAR         VarI;
    GNL_VAR         ActualVarI;
    GNL_FUNCTION   FunctionI;
    GNL_FUNCTION   NewFunction;
    GNL_NODE        NewNode;
    GNL_COMPONENT  ComponentI;
    GNL_COMPONENT  NewCompo;
    BLIST          HashTableNames;
    BLIST          BucketI;
    GNL_NODE       SegmentI;
    GNL            GnlI;

    Interface = GnlUserComponentInterface (UserCompo);
    GnlCompo = GnlUserComponentGnlDef (UserCompo);

    BListQuickDelete (&(GnlInputs (GnlCompo)));
    BListQuickDelete (&(GnlOutputs (GnlCompo)));
    BListQuickDelete (&(GnlLocals (GnlCompo)));
    BListQuickDelete (&(GnlClocks (GnlCompo)));

    /* Create new local var corresponding to the ones in 'GnlCompo' and */
    /* add them in 'Gnl'. Then bind these 'GnlCompo' local vars with the */
    /* new created 'Gnl' local vars.                                         */
    if ((Status = GnlBindAndCreateLocalVar (Gnl, GnlCompo)))
        return (Status);

    /* We bind the actual variables thru the Hook field. This field will */
    /* point on the corresponding formal var.                                */
    if ((Status = GnlBindActualVarOfInterface (Gnl, Interface,
                                                UserCompo)))
        return (Status);

    /* In this case we use the boolean function and replace the actual   */
    /* variables by formal variables.                                         */
    for (i=0; i<BListSize (GnlFunctions (GnlCompo)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (GnlCompo), i);
        ActualVarI = (GNL_VAR)GnlVarHook (VarI);

        FunctionI = GnlVarFunction (VarI);

        /* We do not duplicate ! the function but re-use it and replaces*/
        /* the actual variables by formal variables.                      */
        if (GnlSubstituteNodeForNewGnl (Gnl, GnlFunctionOnSet (FunctionI),
                                         &NewNode))
            return (GNL_MEMORY_FULL);

        free (FunctionI);

        /* If we already created a function for 'ActualVarI' then it is   */
        /* a multi-source signal.                                         */
    }
}

```

```

if (GnlVarFunction (ActualVarI))
{
    fprintf (stderr, " ERROR: multi-source signal for <%s>\n",
             GnlVarName (ActualVarI));
    exit (1);
}

if (GnlFunctionCreate (Gnl, ActualVarI, NewNode, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (ActualVarI, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), ActualVarI))
    return (GNL_MEMORY_FULL);
}
BListQuickDelete (&GnlFunctions (GnlCompo));

/* Now we add the segments of 'GnlCompo' to 'Gnl' since we reuse the */
/* GNL_NODE.                                                 */
/* Actually we will append teh segments of 'Gnl' to the segments of */
/* 'GnlCompo' so that the next call to the creation of GNLNODE will */
/* be done in a 'Gnl' segment. So 'FirstNode' and 'LastNode' keep the*/
/* same.                                                 */
for (i=0; i<BListSize (GnlNodesSegments (Gnl)); i++)
{
    SegmentI = (GNL_NODE)BListElt (GnlNodesSegments (Gnl), i);
    if (BListAddElt (GnlNodesSegments (GnlCompo), (int)SegmentI))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&GnlNodesSegments (Gnl));
SetGnlNodesSegments (Gnl, GnlNodesSegments (GnlCompo));
SetGnlNodesSegments (GnlCompo, NULL);

for (i=0; i<BListSize (GnlComponents (GnlCompo)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (GnlCompo), i);
    if (GnlSubstituteComponentForNewGnl (Gnl, UserCompo, ComponentI,
                                         &NewCompo))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (GnlComponents (Gnl), (int)NewCompo))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&GnlComponents (GnlCompo));

GnlFreeHashNames (GnlCompo);

for (i=0; i<BListSize (GnlHashCompoNames (GnlCompo)); i++)
{
    BucketI = (BLIST)BListElt (GnlHashCompoNames (GnlCompo), i);
    if (BucketI)
        BListQuickDelete (&BucketI);
}
BListQuickDelete (&GnlHashCompoNames (GnlCompo));

/* Finally we remove the current Gnl from the top list 'ListGnls'      */
for (i=0; i<BListSize (ListGnls); i++)
{

```

gnlhier.c

```
GnlI = (GNL)BListElt (ListGnls, i);
if (GnlI == GnlCompo)
{
    BListDelInsert (ListGnls, i+1);
    break;
}
}

free (GnlName (GnlCompo));
free ((char*)GnlCompo);

return (GNL_OK);
}

/*
 * GnlReplaceDuplicateComponentInGnl
 */
/*
 * This procedure makes a copy of the gnl corresponding to 'UserCompo'  */
/* and adds this new gnl in the gnl 'Gnl'.                                */
/*
GNL_STATUS GnlReplaceDuplicateComponentInGnl (Gnl, UserCompo)
    GNL             Gnl;
    GNL_USER_COMPONENT UserCompo;
{
    BLIST           Interface;
    GNL_STATUS      Status;
    GNL             GnlCompo;
    int             i;
    GNL_VAR         VarI;
    GNL_VAR         ActualVarI;
    GNL_FUNCTION    FunctionI;
    GNL_FUNCTION    NewFunction;
    GNL_NODE        NewNode;
    GNL_COMPONENT   ComponentI;
    GNL_COMPONENT   NewCompo;

    Interface = GnlUserComponentInterface (UserCompo);
    GnlCompo = GnlUserComponentGnlDef (UserCompo);

    /* Create new local var corresponding to the ones in 'GnlCompo' and */
    /* add them in 'Gnl'. Then bind these 'GnlCompo' local vars with the */
    /* new created 'Gnl' local vars.                                         */
    if ((Status = GnlBindAndCreateLocalVar (Gnl, GnlCompo)))
        return (Status);

    /* We bind the actual variables thru the Hook field. This field will */
    /* point on the corresponding formal var.                               */
    if ((Status = GnlBindActualVarOfInterface (Gnl, Interface,
                                                UserCompo)))
        return (Status);

    /* First we duplicate the boolean function and replace the actual      */
    /* variables by formal variables.                                       */
    for (i=0; i<BListSize (GnlFunctions (GnlCompo)); i++)
    {
```

```

VarI = (GNL_VAR)BListElt (GnlFunctions (GnlCompo), i);
ActualVarI = (GNL_VAR)GnlVarHook (VarI);

FunctionI = GnlVarFunction (VarI);

if (GnlDuplicateNodeForNewGnl (Gnl, GnlFunctionOnSet (FunctionI),
                               &NewNode))
    return (GNL_MEMORY_FULL);

if (GnlFunctionCreate (Gnl, ActualVarI, NewNode, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (ActualVarI, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), ActualVarI))
    return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (GnlComponents (GnlCompo)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (GnlCompo), i);

    if (GnlDuplicateComponentForNewGnl (Gnl, UserCompo,
                                         ComponentI, &NewCompo))
        return (GNL_MEMORY_FULL);

    if (BListAddElt (GnlComponents (Gnl), (int)NewCompo))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 *-----*
 * GnlFlattenFullGnlComponentsRec
 *-----*/
/* This procedure flattens all the User components of 'Gnl' (and */
/* recursively the components of the components) and replace them by */
/* their corresponding Gnl representant. */
/* This procedure makes a side effect on the list 'ListGnls' by removing*/
/* the element 'Gnl' present in 'ListGnls'. */
/*-----*/
GNL_STATUS GnlFlattenFullGnlComponentsRec (Nw, Gnl, ListGnls)
    GNL_NETWORK     Nw;
    GNL           Gnl;
    BLIST      ListGnls;
{
    BLIST          Components;
    int            i;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL_STATUS      Status;
    GNL           GnlCompoI;
    BLIST      ListFlatten;
    int            ModulePresent;
}

```

```

Components = GnlComponents (Gnl);
if (!Components)
    return (GNL_OK);

/* If the 'Gnl' has been already processed it is not necessary to      */
/* re-flatten its user components                                         */
if (GnlTag (Gnl) == GnlNetworkTag (Nw))
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    /* If we did not find any Gnl definition then it is a black box */
    /* that we preserve.                                              */
    if (!GnlCompoI)
        continue;

    /* Doing recursively the flattening of the Gnl definition of      */
    /* 'UserCompoI'.                                                 */
    if ((Status = GnlFlattenFullGnlComponentsRec (Nw, GnlCompoI,
                                                   ListGnls)))
        return (Status);

    /* we are doing selective flattening ...                         */
    if ((GnlEnvFlatten () == GNL_EXCEPT_FLATTEN) ||
        (GnlEnvFlatten () == GNL_ONLY_FLATTEN))
    {
        ListFlatten = GnlEnvFlatListModules ();
        ModulePresent = BListMemberOfList (ListFlatten,
                                           GnlName (GnlCompoI), StringIdentical);
        if ((ModulePresent &&
             (GnlEnvFlatten () == GNL_EXCEPT_FLATTEN)) ||
            (!ModulePresent &&
             (GnlEnvFlatten () == GNL_ONLY_FLATTEN)))
            continue;
    }

    /* If the user wants to flatten the leaf cells.                  */
    if (GnlEnvFlatten () == GNL_LEAF_FLATTEN)
    {
        /* If the Gnl of the component is not of type GNL_CELL then */
        /* we do not flatten it.                                     */
        if (GnlType (GnlCompoI) != GNL_CELL)
            continue;
    }
}

```

```

BListDelInsert (Components, i+1);
i--;

if (GnlRefCount (GnlCompoI) == 1)
{
    /* This is the last Gnl to collapse so we take it directly      */
    /* instead of duplicating it and freeing it afterwards. */
    if ((Status = GnlReplaceSubstituteComponentInGnl (Gnl,
                                                       UserCompoI, ListGnls)))
        return (Status);
}
else
{
    /* There are still some replacements to perform. We replace*/
    /* physically the component by its Gnl definition duplicate*/
    if ((Status = GnlReplaceDuplicateComponentInGnl (Gnl,
                                                       UserCompoI)))
        return (Status);
}

SetGnlRefCount (GnlCompoI, GnlRefCount (GnlCompoI)-1);

GnlFreeComponent (ComponentI);
}

return (GNL_OK);
}

/*
 * GnlUpdateLocalVarDirSeqCompo
 */
/*
 * Updates the Direction of the ports of the Sequential component
 * 'SeqCompo'.
 */
void GnlUpdateLocalVarDirSeqCompo (SeqCompo)
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
{

if (!GnlVarIsPrimary (GnlSequentialCompoInput (SeqCompo)))
    SetGnlVarDir (GnlSequentialCompoInput (SeqCompo),
                  GNL_VAR_LOCAL_WIRING);

if (GnlSequentialCompoOutput (SeqCompo) &&
    !GnlVarIsPrimary (GnlSequentialCompoOutput (SeqCompo)))
    SetGnlVarDir (GnlSequentialCompoOutput (SeqCompo),
                  GNL_VAR_LOCAL_WIRING);

if (GnlSequentialCompoOutputBar (SeqCompo) &&
    !GnlVarIsPrimary (GnlSequentialCompoOutputBar (SeqCompo)))
    SetGnlVarDir (GnlSequentialCompoOutputBar (SeqCompo),
                  GNL_VAR_LOCAL_WIRING);

if (!GnlVarIsPrimary (GnlSequentialCompoClock (SeqCompo)))
    SetGnlVarDir (GnlSequentialCompoClock (SeqCompo),
                  GNL_VAR_LOCAL_WIRING);
}

```

gnlhier.c

```
if (GnlSequentialCompoReset (SeqCompo) &&
    !GnlVarIsPrimary (GnlSequentialCompoReset (SeqCompo)))
SetGnlVarDir (GnlSequentialCompoReset (SeqCompo),
              GNL_VAR_LOCAL_WIRING);

if (GnlSequentialCompoSet (SeqCompo) &&
    !GnlVarIsPrimary (GnlSequentialCompoSet (SeqCompo)))
SetGnlVarDir (GnlSequentialCompoSet (SeqCompo),
              GNL_VAR_LOCAL_WIRING);

}

/*-----
/* GnlUpdateLocalVarDirTristateCompo
 */
/*-----*/
/* Updates the Direction of the ports of the tri state 'TriCompo'. */
/*-----*/
void GnlUpdateLocalVarDirTristateCompo (TriCompo)
    GNL_TRISTATE_COMPONENT     TriCompo;
{

    if (!GnlVarIsPrimary (GnlTriStateInput (TriCompo)))
        SetGnlVarDir (GnlTriStateInput (TriCompo),
                      GNL_VAR_LOCAL_WIRING);

    if (!GnlVarIsPrimary (GnlTriStateOutput (TriCompo)))
        SetGnlVarDir (GnlTriStateOutput (TriCompo),
                      GNL_VAR_LOCAL_WIRING);

    if (!GnlVarIsPrimary (GnlTriStateSelect (TriCompo)))
        SetGnlVarDir (GnlTriStateSelect (TriCompo),
                      GNL_VAR_LOCAL_WIRING);
}

/*-----
/* GnlUpdateLocalVarDirBufCompo
 */
/*-----*/
/* Updates the Direction of the ports of the buffer 'BufCompo'. */
/*-----*/
void GnlUpdateLocalVarDirBufCompo (BufCompo)
    GNL_BUF_COMPONENT         BufCompo;
{

    if (!GnlVarIsPrimary (GnlBufInput (BufCompo)))
        SetGnlVarDir (GnlBufInput (BufCompo),
                      GNL_VAR_LOCAL_WIRING);

    if (!GnlVarIsPrimary (GnlBufOutput (BufCompo)))
        SetGnlVarDir (GnlBufOutput (BufCompo),
                      GNL_VAR_LOCAL_WIRING);
}

/*-----
/* GnlUpdateLocalVarDirUserCompo
 */
/*-----*/
```

```

/* Scans the interface of 'UserCompoI' and updates the direction of each*/
/* actual variable by giving the GNL_VAR_LOCAL_WIRING direction. */
/*-----*/
GNL_STATUS GnlUpdateLocalVarDirUserCompo (Gnl, UserCompoI)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        UserCompoI;
{
    int                     j;
    BLIST      Interface;
    int                     k;
    BLIST      ListSplitActuals;
    GNL_VAR      VarK;
    GNL_ASSOC     AssocJ;
    GNL_VAR      Actual;
    GNL_STATUS    GnlStatus;
    int                     i;
    int                     LeftActIndex;
    int                     RightActIndex;
    char                  *IndexActualName;
    GNL_VAR      IndexActualVar;

    Interface = GnlUserComponentInterface (UserCompoI);

    for (j=0; j<BListSize (Interface); j++)
    {
        AssocJ = (GNL_ASSOC)BListElt (Interface, j);

        Actual = GnlAssocActualPort (AssocJ);

        /* case of an open port. */
        if (!Actual)
            continue;

        if (GnlVarIsVar (Actual))
        {
            if (GnlVarRangeSize (Actual) == 1)
            {
                if (!GnlVarIsPrimary (Actual))
                    SetGnlVarDir (Actual, GNL_VAR_LOCAL_WIRING);
            continue;
            }

            /* if not it is a bus and we need also to change the */
            /* direction of the expanded bit variables associated to it */
            LeftActIndex = GnlVarMsb (Actual);
            RightActIndex = GnlVarLsb (Actual);
            if (LeftActIndex > RightActIndex)
            {
                LeftActIndex = GnlVarLsb (Actual);
                RightActIndex = GnlVarMsb (Actual);
            }
            for (i=LeftActIndex; i<=RightActIndex; i++)
            {
                if (GnlVarIndexName (Actual, i, &IndexActualName))
                    return (GNL_MEMORY_FULL);
                if ((GnlStatus = GnlGetVarFromName (Gnl,

```

```

                IndexActualName, &IndexActualVar)))
{
    if (GnlStatus == GNL_VAR_NOT_EXISTS)
        continue;
    return (GNL_MEMORY_FULL);
}
free (IndexActualName);
if (!GnlVarIsPrimary (IndexActualVar))
    SetGnlVarDir (IndexActualVar, GNL_VAR_LOCAL_WIRING);
}
else
{
/* This is a CONCAT_OP with a list of Vars. */
ListSplitActuals = GnlNodeSons ((GNL_NODE)Actual);
for (k=0; k<BListSize (ListSplitActuals); k++)
{
    VarK = (GNL_VAR)BListElt (ListSplitActuals, k);
    if (!GnlVarIsPrimary (VarK))
        SetGnlVarDir (VarK, GNL_VAR_LOCAL_WIRING);
}
}
}

return (GNL_OK);
}

/*
 * GnlUpdateVarDir
 */
/*
 * Some Var directions may have changed after the full flattening phase */
/* For instance an original GNL_VAR_LOCAL_WIRING variable may become a */
/* simple GNL_VAR_LOCAL. This procedure updates then the field GnlVarDir */
/* of each var. GnlVarDir is GNL_VAR_LOCAL_WIRING only if this var */
/* appears in one of the components interfaces. */
*/
GNL_STATUS GnlUpdateLocalVarDirRec (Nw, Gnl)
    GNL_NETWORK      Nw;
    GNL             Gnl;
{
    int                  i;
    BLIST            BucketI;
    int                  j;
    GNL_VAR           VarJ;
    GNL_COMPONENT     ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    BLIST            Components;
    BLIST            HashTableNames;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TriCompo;
    GNL_BUF_COMPONENT BufCompo;
    GNL               GnlCompoI;

    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);
}

```

```

SetGnlTag (Gnl, GnlNetworkTag (Nw)) ;

/* First we change all the non IOs signals (INPUT,OUTPUT,INOUT), into*/
/* GNL_VAR_LOCAL. This means we modify the GNL_VAR_LOCAL_WIRING      */
/* signals into GNL_VAR_LOCAL.                                         */
HashTableNames = GnlHashNames (Gnl) ;
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);

        if (!GnlVarIsIO (VarJ))
            SetGnlVarDir (VarJ, GNL_VAR_LOCAL) ;
    }
}

Components = GnlComponents (Gnl) ;
if (!Components)
    return (GNL_OK) ;

/* Now we scan the components in order to re-update the variables      */
/* which are still GNL_VAR_LOCAL_WIRING.                                */
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    switch (GnlComponentType (ComponentI)) {
        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
            GnlUpdateLocalVarDirSeqCompo (SeqCompo);
            break;

        case GNL_TRISTATE_COMPO:
            TriCompo = (GNL_TRISTATE_COMPONENT)ComponentI;
            GnlUpdateLocalVarDirTristateCompo (TriCompo);
            break;

        case GNL_BUF_COMPO:
            BufCompo = (GNL_BUF_COMPONENT)ComponentI;
            GnlUpdateLocalVarDirBufCompo (BufCompo);
            break;

        case GNL_MACRO_COMPO:
            break;

        case GNL_USER_COMPO:
            UserCompoI = (GNL_USER_COMPONENT)ComponentI;
            if (GnlUpdateLocalVarDirUserCompo (Gnl, UserCompoI))
                return (GNL_MEMORY_FULL);

            GnlCompoI = GnlUserComponentGnlDef (UserCompoI);
            if (GnlCompoI)
            {
                if (GnlUpdateLocalVarDirRec (Nw, GnlCompoI))
                    return (GNL_MEMORY_FULL);
            }
    }
}

```

```

        }

        break;
    }

}

return (GNL_OK);
}

/*
 * GnlFlattenFullGnlComponents
 */
/*
 * This procedure flattens all the User components of 'Gnl' (and
 * recursively the components of the components) and replace them by
 * their corresponding Gnl representant.
 */
/* The flattening can be limited by User Options which are:
 */
/*      GnlEnvFlatten () = GNL_EXCEPT_FLATTEN
 */
/*      GnlEnvFlatten () = GNL_ONLY_FLATTEN
 */
/*
 * This procedure makes a side effect on the list 'ListGnls' by removing
 * the Gnl elements which will be completely flattened (e.g not used
 * anywhere).
 */
/*
 *-----*
GNL_STATUS GnlFlattenFullGnlComponents (Nw, Gnl, ListGnls)
{
    GNL_NETWORK    Nw;
    GNL           Gnl;
    BLIST          ListGnls;

    int            i;
    GNL_VAR        VarI;
    GNL_FUNCTION   Function;
    GNL_NODE       Node;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    if (GnlFlattenFullGnlComponentsRec (Nw, Gnl, ListGnls))
        return (GNL_MEMORY_FULL);

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    /* Updates the direction of local variables after the collapsing
     * phase. Indeed, a GNL_LOCAL_WIRING can become a GNL_LOCAL since it
     * does not drive a USER component anymore.
     */
    if (GnlUpdateLocalVarDirRec (Nw, Gnl))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
 *-----* EOF -----*

```

gnlinjct.c

```
/*-----*/  
/*-----*/  
/*      File:          gnlinjct.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Description: */  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY           /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
  
#include "blist.e"  
  
/*-----*/  
/* GnlNodeNbLitt */  
/*-----*/  
int GnlNodeNbLitt (Node)  
    GNL_NODE Node;  
{  
    int             i;  
    int             NbLitt;  
    GNL_NODE SonI;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
        return (0);  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
        return (1);  
  
    NbLitt = 0;  
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
    {  
        SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);  
        NbLitt += GnlNodeNbLitt (SonI);  
    }  
  
    return (NbLitt);  
}
```

gnlinjct.c

```
/*-----*/
/* GnlVarNbOccurInNode */
/*-----*/
static      BLIST G_ListFunctions;
int GnlVarNbOccurInNode (Node, Var)
    GNL_NODE Node;
    GNL_VAR  Var;
{
    int             i;
    GNL_VAR  VarSon;
    int             NbOccur;
    BLIST   Sons;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (0);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        VarSon = (GNL_VAR) GnlNodeSons (Node);
        if (Var == VarSon)
            return (1);
        return (0);
    }

    NbOccur = 0;
    Sons = GnlNodeSons (Node);
    for (i=0; i<BListSize (Sons); i++)
    {
        SonI = (GNL_NODE) BlistElt (Sons, i);
        NbOccur += GnlVarNbOccurInNode (SonI, Var);
    }

    return (NbOccur);
}

/*-----*/
/* GnlVarNbOccur */
/*-----*/
GNL_STATUS GnlVarNbOccur (Gnl, Var, MaxOccur, ListFunctions, NbOccur)
    GNL           Gnl;
    GNL_VAR  Var;
    float        MaxOccur;
    BLIST   *ListFunctions;
    int            *NbOccur;
{
    int             i;
    GNL_VAR  VarI;
    GNL_FUNCTION  FunctionI;
    int            NbOccurI;

    if (BListCreate (&G_ListFunctions))
        return (GNL_MEMORY_FULL);
```

gnlinjct.c

```
*NbOccur = 0;

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    NbOccurI = GnlVarNbOccurInNode (GnlFunctionOnSet (FunctionI),
                                    Var);
    if (NbOccurI)
    {
        *NbOccur += NbOccurI;
        if (BListAddElt (G_ListFunctions, (int)i))
            return (GNL_MEMORY_FULL);

        if (MaxOccur < (float)(*NbOccur))
        {
            *NbOccur = 0;
            *ListFunctions = G_ListFunctions;
            return (GNL_OK);
        }
    }
}

*ListFunctions = G_ListFunctions;
return (GNL_OK);
}

/*-----
/* GnlIncrementDadsFromNode
*/
void GnlIncrementDadsFromNode (Node)
    GNL_NODE Node;
{
    GNL_VAR Var;
    int i;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        SetGnlVarDads (Var, (BLIST)((int)GnlVarDads(Var)+1));
        return;
    }

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        GnlIncrementDadsFromNode (SonI);
    }
}

/*-----*/
```

gnlinjct.c

```
/* GnlInjectNodeInNode */  
/*-----*/  
/* Scan recursively node 'Node1' in order to replace occurence of 'Var' */  
/* by 'Node2'. */  
/*-----*/  
void GnlInjectNodeInNode (Node1, Var, Node2, NewNode, NbOccur)  
{  
    GNL_NODE      Node1;  
    GNL_VAR       Var;  
    GNL_NODE      Node2;  
    GNL_NODE      *NewNode;  
    int           *NbOccur;  
    {  
        int          i;  
        GNL_NODE     SonI;  
  
        if (GnlNodeOp (Node1) == GNL_VARIABLE)  
        {  
            if (Var == (GNL_VAR) GnlNodeSons (Node1))  
            {  
                (*NbOccur)--;  
                *NewNode = Node2;  
                /* Since this node is substituted, all its expression is */  
                /* used one more time. */  
                GnlIncrementDadsFromNode (*NewNode);  
                return;  
            }  
            *NewNode = Node1;  
            return;  
        }  
        if (GnlNodeOp (Node1) == GNL_CONSTANTE)  
        {  
            *NewNode = Node1;  
            return;  
        }  
        for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)  
        {  
            SonI = (GNL_NODE) BListElt (GnlNodeSons (Node1), i);  
            GnlInjectNodeInNode (SonI, Var, Node2, NewNode, NbOccur);  
            BListElt (GnlNodeSons (Node1), i) = (int) (*NewNode);  
        }  
        *NewNode = Node1;  
    }  
    /*-----*/  
    /* GnlDecrementNumberDadsFromNode */  
    /*-----*/  
void GnlDecrementNumberDadsFromNode (Node)  
{  
    GNL_NODE      Node;  
    GNL_VAR       Var;  
    int           i;  
    GNL_NODE     SonI;
```

```

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    Var = (GNL_VAR)GnlNodeSons (Node);
    SetGnlVarDads (Var, (BLIST)((int)GnlVarDads(Var)-1));
    return;
}

if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return;

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    GnlDecrementNumberDadsFromNode (SonI);
}
}

/*-----*/
/* GnlInjectLocalVar */
/*-----*/
void GnlInjectLocalVar (Gnl, Function, NbOccur)
{
    GNL          Gnl;
    GNL_FUNCTION Function;
    int          *NbOccur;
{
    GNL_VAR  Var;
    GNL_NODE NodeExpression;
    int      i;
    GNL_VAR  VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE NodeI;
    GNL_NODE NewNode;
    int      IndexFunction;

    Var = GnlFunctionVar (Function);
    NodeExpression = GnlFunctionOnSet (Function);

    /* We decrement of one use each var in this expression that we will  */
    /* substitute.                                                       */
    GnlDecrementNumberDadsFromNode (NodeExpression);

    for (i=0; i<BListSize (GnlFunctions(Gnl)); i++)
    {
        if (!(*NbOccur))
            return;
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        NodeI = GnlFunctionOnSet (FunctionI);

        GnlInjectNodeInNode (NodeI, Var, NodeExpression, &NewNode,
                           NbOccur);
        SetGnlFunctionOnSet (FunctionI, NewNode);
    }
}

```

gnlinjct.c

```
}

/*-----*/
/* GnlResetDadsInVar */
/*-----*/
/* This procedure scans all the variables of 'Gnl' thru the hash table */
/* names and reset to NULL the field 'GnlVarDads' of each var. */
/*-----*/
void GnlResetDadsInVar (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR    VarJ;
    BLIST     HashTableNames;
    BLIST     BucketI;
    int          j;

    HashTableNames = GnlHashNames (Gnl);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST)BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);
            SetGnlVarDads (VarJ, NULL);
        }
    }
}

/*-----*/
/* GnlGetNumberDadsFromGnl */
/*-----*/
/* This procedure uses the Field 'GnlVarDads' of each variables of the */
/* 'Gnl'. It uses as in Integer to store number of times a Var is refe- */
/* rened (e.g number of dads). */
/*-----*/
void GnlGetNumberDadsFromGnl (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR    VarI;
    GNL_NODE NodeI;
    GNL_FUNCTION FunctionI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);

        FunctionI = GnlVarFunction (VarI);
        if (!FunctionI)
            continue;
```

gnlinjct.c

```
NodeI = GnlFunctionOnSet (FunctionI);

    GnlIncrementDadsFromNode (NodeI);
}
}

/*-----*/
/* GnlGetDadsFromNode */
/*-----*/
GNL_STATUS GnlGetDadsFromNode (Node, Dad)
    GNL_NODE Node;
    GNL_NODE Dad;
{
    GNL_VAR Var;
    BLIST NewList;
    int i;
    GNL_NODE SonI;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (Node);
        if (GnlVarDir (Var) != GNL_VAR_LOCAL)
            return (GNL_OK);

        if (GnlVarIsVss (Var))
            return (GNL_OK);

        if (GnlVarIsVdd (Var))
            return (GNL_OK);

        if (GnlVarDads (Var) == NULL)
        {
            if (BListCreateWithSize (1, &NewList))
                return (GNL_MEMORY_FULL);
            SetGnlVarDads (Var, NewList);
        }
        if (BListAddElt (GnlVarDads (Var), (int) Dad))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
        if (GnlGetDadsFromNode (SonI, Node))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlGetDadsFromGnl */
/*-----*/
```

gnlinjct.c

```
/*
 *-----*
 * This procedure computes for each var the list of its fathers. */
 *-----*/
GNL_STATUS GnlGetDadsFromGnl (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR    VarI;
    GNL_NODE  NodeI;
    GNL_FUNCTION FunctionI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);

        FunctionI = GnlVarFunction (VarI);
        if (FunctionI == NULL)
            continue;

        NodeI = GnlFunctionOnSet (FunctionI);

        if (GnlGetDadsFromNode (NodeI, VarI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 *-----*
 /* GnlInjectQuick
 *-----*/
/* This procedure re-injects local Funtions such that the relation
 * below is verified:
 */
/*      F reinjected <=> (NbOccur(F) * (NbLitt(F)-1) - NbLitt(F) < Limit) */
/*
 *-----*/
GNL_STATUS GnlInjectQuick (Gnl, Limit)
    GNL      Gnl;
    int      Limit;
{
    int          i;
    int          NbLitt;
    int          NbOccur;
    GNL_FUNCTION FunctionI;
    GNL_VAR    VarI;
    float     MaxOccur;
    int          j;
    int          k;
    GNL_NODE  DadJ;
    GNL_FUNCTION FunctionJ;
    GNL_NODE  NodeK;
    GNL_NODE  NodeI;
    BLIST     NewList;
    int          MaxNbLitt;
```

```

/* we sort the functions from the deepest to the highest ones. */
GnlSortFunctionsOnInvertLevel (Gnl);

GnlResetDadsInVar (Gnl);

/* we compute the Dads for each Variable. */
GnlGetDadsFromGnl (Gnl);

/* We process first the deepest functions and go up in the circuit. */
/* This is usefull so that by reinjecting a sub-function we do not */
/* change the list of the fathers of the higher vars. */
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    fprintf (stderr, "%c Reinjecting glue logic (size = %d) [%d/%d]",
            13, Limit, i,
            BListSize (GnlFunctions (Gnl)));
    fflush (stderr);

VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
FunctionI = GnlVarFunction (VarI);

/* we substitute only LOCAL variables */
if (GnlVarDir (VarI) != GNL_VAR_LOCAL)
    continue;

NbOccur = BListSize (GnlVarDads (VarI));

/* We compute a necessary condition to verify the condition. */
/* If the necessary condition is not true then it is not */
/* necessary to continue. */
if (NbOccur != 1)
{
    MaxNbLitt = (Limit-NbOccur)/(NbOccur-1);
    if (MaxNbLitt < 1)
    {
        BListQuickDelete (&GnlVarDads (VarI));
        SetGnlVarDads (VarI, NULL);
        continue;
    }
}

NbLitt = GnlNodeNbLitt (GnlFunctionOnSet (FunctionI));

if (NbOccur && ((NbOccur*(NbLitt -1)-NbLitt) < Limit))
{
    NodeI = GnlFunctionOnSet (FunctionI);

    /* We replace all occurrences of 'VarI' by its expression */
    /* NodeI. This variable 'VarI' is then removed from the Gnl. */
    for (j=0; j<BListSize (GnlVarDads (VarI)); j++)
    {
        DadJ = (GNL_NODE)BListElt (GnlVarDads (VarI), j);
        if (GnlVarIsVar ((GNL_VAR)DadJ))
        {

```

```

/* Actually 'DadJ' is a GNL_VAR */  

FunctionJ = GnlVarFunction ((GNL_VAR) DadJ);  

SetGnlFunctionOnSet (FunctionJ, NodeI);  

}  

else  

{  

    for (k=0; k<BListSize (GnlNodeSons (DadJ)); k++)  

    {  

        NodeK = (GNL_NODE) BListElt (GnlNodeSons (DadJ),  

                                      k);  

        if ((GnlNodeOp (NodeK) == GNL_VARIABLE) &&  

            (VarI == (GNL_VAR) GnlNodeSons (NodeK)))  

        {  

            BListElt (GnlNodeSons (DadJ), k) = (int)  

                NodeI;  

        }
    }
}  

}  

BListQuickDelete (&GnlVarDads (VarI));  

SetGnlVarDads (VarI, NULL);  

/* We remove 'VarI' from the list locals of 'Gnl'. */  

GnlRemoveVarFromGnlLocals (Gnl, VarI);  

BListElt (GnlFunctions (Gnl), i) = (int) NULL;  

SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)-1);  

/* We remove 'VarI' from the hash table names of 'Gnl'. */  

GnlRemoveVarFromGnlHashTableName (Gnl, VarI);  

GnlFunctionFree (GnlVarFunction (VarI));  

GnlFreeVar (VarI);  

}  

else if (GnlVarDads (VarI))  

{  

    BListQuickDelete (&GnlVarDads (VarI));  

    SetGnlVarDads (VarI, NULL);
}  

}  

if (BListSize (GnlFunctions (Gnl)))  

{
    fprintf (stderr, "%c Reinjecting glue logic (size = %d) [%d/%d] ",  

            13, Limit, i, BListSize (GnlFunctions (Gnl)));  

    fflush (stderr);
    fprintf (stderr, "\n");
}  

if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);  

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    if (BListElt (GnlFunctions (Gnl), i))
    {
        if (BListAddElt (NewList,

```

```

        BListElt (GnlFunctions (Gnl), i)))
    return (GNL_MEMORY_FULL);
}
}

BListQuickDelete (&GnlFunctions (Gnl));
SetGnlFunctions (Gnl, NewList);

GnlResetDadsInVar (Gnl);

return (GNL_OK);
}

/*-----*/
/* GnlInject */
/*-----*/
/* This procedure re-injects local Functions such that the relation */
/* below is verified: */
/*
/*      F reinjected <=> (NbOccur(F) * (NbLitt(F)-1) - NbLitt(F) < Limit) */
/*-----*/
GNL_STATUS GnlInject (Gnl, Limit)
    GNL        Gnl;
    int        Limit;
{
    int            i;
    int            NbLitt;
    int            NbOccur;
    GNL_FUNCTION  FunctionI;
    GNL_VAR       Vari;
    float         MaxOccur;

    /* we sort the functions from the deepest to the highest ones. */
    GnlSortFunctionsOnLevel (Gnl);

    GnlResetDadsInVar (Gnl);

    /* we compute the number of Dads for Variable. */
    GnlGetNumberDadsFromGnl (Gnl);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (Vari);
        if (GnlVarDir (Vari) != GNL_VAR_LOCAL)
            continue;

        /* we substitute only LOCAL variables */
        NbLitt = GnlNodeNbLitt (GnlFunctionOnSet (FunctionI));

        NbOccur = (int)GnlVarDads (Vari);

        if (NbOccur && ((NbOccur*(NbLitt -1)-NbLitt) < Limit))
        {
            /* We replace all occurrences of 'Vari' by its expression */
            /* Node. This variable 'Vari' is then removed from the Gnl. */
        }
    }
}

```

```

        GnlInjectLocalVar (Gnl, FunctionI, &NbOccur);

        BListDelShift (GnlFunctions (Gnl), i+1);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)-1);

        GnlRemoveVarFromGnlLocals (Gnl, VariI);
        i--;
    }

}

GnlResetDadsInVar (Gnl);

return (GNL_OK);
}

/*
/* GnlGetSingleVarNode
*/
void GnlGetSingleVarNode (Node1, Node2)
    GNL_NODE Node1;
    GNL_NODE *Node2;
{
    GNL_VAR Var;
    int i;
    GNL_NODE NodeI;
    GNL_NODE NewNodeI;

    if (GnlNodeOp (Node1) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (Node1);
        if (GnlVarDir (Var) != GNL_VAR_LOCAL)
        {
            *Node2 = Node1;
            return;
        }
        if (GnlVarDads (Var) != (BLIST)1)
        {
            *Node2 = Node1;
            return;
        }
        if (GnlVarFunction (Var) == NULL)
        {
            *Node2 = Node1;
            return;
        }
        *Node2 = GnlFunctionOnSet (GnlVarFunction (Var));
        /* if extra info stored on this node we do not collapse it */
        if (GnlNodeHook (*Node2) && (GnlMapNodeInfoOriginalCompo (*Node2)))
        {
            *Node2 = Node1;
            return;
        }
    }
}

```

gnlinject.c

```
        return;
    }

    if (GnlNodeOp (Node1) == GNL_CONSTANTE)
    {
        *Node2 = Node1;
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)
    {
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node1), i);
        GnlGetSingleVarNode (NodeI, &NewNodeI);
        BListElt (GnlNodeSons (Node1), i) = (int)NewNodeI;
    }

    *Node2 = Node1;
}

/*-----*/
/* GnlGetSingleVarNodeWithMaxLitt */
/*-----*/
void GnlGetSingleVarNodeWithMaxLitt (Node1, Node2, MaxLitt)
{
    GNL_NODE Node1;
    GNL_NODE *Node2;
    int          MaxLitt;
{
    GNL_VAR  Var;
    int          i;
    GNL_NODE NodeI;
    GNL_NODE NewNodeI;

    if (GnlNodeOp (Node1) == GNL_CONSTANTE)
    {
        *Node2 = Node1;
        return;
    }

    if (GnlNodeOp (Node1) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node1);
        if (GnlVarDir (Var) != GNL_VAR_LOCAL)
        {
            *Node2 = Node1;
            return;
        }
        if (GnlVarDads (Var) != (BLIST)1)
        {
            *Node2 = Node1;
            return;
        }
        if (GnlVarFunction (Var) == NULL)
        {
            *Node2 = Node1;
            return;
        }
    }
}
```

gnlinject.c

```
        }
        if (GnlNodeNbLitt (GnlFunctionOnSet (
            GnlVarFunction (Var))) >= MaxLitt)
    {
        *Node2 = Node1;
        return;
    }

    *Node2 = GnlFunctionOnSet (GnlVarFunction (Var));
    return;
}

for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)
{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node1), i);
    GnlGetSingleVarNodeWithMaxLitt (NodeI, &NewNodeI, MaxLitt);
    BListElt (GnlNodeSons (Node1), i) = (int)NewNodeI;
}

*Node2 = Node1;
}

/*-----*/
/* GnlInjectVarWithUniqueDad                         */
/*-----*/
void GnlInjectVarWithUniqueDad (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_NODE NewNode;
    GNL_VAR   Vari;
    GNL_FUNCTION FunctionI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (Vari);
        GnlGetSingleVarNode (GnlFunctionOnSet (FunctionI), &NewNode);
        SetGnlFunctionOnSet (FunctionI, NewNode);
    }
}

/*-----*/
/* GnlInjectVarWithUniqueDadWithMaxLitt             */
/*-----*/
void GnlInjectVarWithUniqueDadWithMaxLitt (Gnl, MaxLitt)
    GNL      Gnl;
    int      MaxLitt;
{
    int          i;
    GNL_NODE NewNode;
    GNL_VAR   Vari;
    GNL_FUNCTION FunctionI;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
```

gnlinjct.c

```
{  
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);  
    FunctionI = GnlVarFunction (Vari);  
    GnlGetSingleVarNodeWithMaxLitt (GnlFunctionOnSet (FunctionI),  
                                    &NewNode, MaxLitt);  
    SetGnlFunctionOnSet (FunctionI, NewNode);  
}  
}  
  
/*-----*/  
/* GnlRemoveVarFromLocals */  
/*-----*/  
void GnlRemoveVarFromLocals (Gnl, Var)  
{  
    GNL Gnl;  
    GNL_VAR Var;  
    int i;  
  
    for (i=0; i<BListSize (GnlLocals (Gnl)); i++)  
    {  
        if (Var == (GNL_VAR)BListElt (GnlLocals (Gnl), i))  
        {  
            BListDelInsert (GnlLocals (Gnl), i+1);  
            SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)-1);  
            break;  
        }  
    }  
}  
  
/*-----*/  
/* GnlRemoveUnusedVar */  
/*-----*/  
GNL_STATUS GnlRemoveUnusedVar (Gnl)  
{  
    GNL Gnl;  
    int i;  
    GNL_VAR VarJ;  
    GNL_VAR Vari;  
    BLIST NewList;  
    BLIST HashTableNames;  
    BLIST BucketI;  
    int j;  
  
    GnlResetDadsInVar (Gnl);  
  
    /* we compute the number of Dads for Variable. */  
    GnlGetNumberDadsFromGnl (Gnl);  
  
    /* We reset the number of locals and recompute everything */  
    BListQuickDelete (&GnlLocals (Gnl));  
    SetGnlNbLocal (Gnl, 0);  
  
    if (BListCreate (&NewList))
```

gnlinjct.c

```
    return (GNL_MEMORY_FULL);
SetGnlLocals (Gnl, NewList);

/* Removing the unused functions.                                         */
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    if (GnlVarDir (VarI) != GNL_VAR_LOCAL)
        continue;

    if (GnlVarDads (VarI) == 0)
    {
        BListDelInsert (GnlFunctions (Gnl), i+1);
        i--;
    }
}

/* Removing the unused GNL_VAR_LOCAL variables                         */
HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        switch (GnlVarDir (VarJ)) {
        case GNL_VAR_LOCAL:
            if (GnlVarDads (VarJ) == 0)
            {
                /* we do not free the buses.                      */
                if (GnlVarRangeSize (VarJ) == 1)
                {
                    BListDelInsert (BucketI, j+1);
                    GnlFreeVar (VarJ);
                    j--;
                }
            }
            else
            {
                SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
                if (BListAddElt (NewList, (int)VarJ))
                    return (GNL_MEMORY_FULL);
            }
            break;

        case GNL_VAR_LOCAL_WIRING:
            SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
            if (BListAddElt (NewList, (int)VarJ))
                return (GNL_MEMORY_FULL);
            break;
        default:
            break;
        }
    }
}
```

gnlinjct.c

```
    return (GNL_OK);

}

/*-----*/
/* GnlInjectAllVarInNode */
/*-----*/
void GnlInjectAllVarInNode (Node1, NewNode)
    GNL_NODE      Node1;
    GNL_NODE      *NewNode;
{
    int          i;
    GNL_NODE     SonI;
    GNL_FUNCTION  Function;
    GNL_VAR      Var;

    if (GnlNodeOp (Node1) == GNL_CONSTANTE)
    {
        *NewNode = Node1;
        return;
    }

    if (GnlNodeOp (Node1) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node1);
        if ((GnlVarDir (Var) == GNL_VAR_INPUT) ||
            (GnlVarDir (Var) == GNL_VAR_LOCAL_WIRING))
        {
            *NewNode = Node1;
            return;
        }
        if (GnlVarFunction (Var) == NULL)
        {
            *NewNode = Node1;
            return;
        }
        Function = GnlVarFunction (Var);
        *NewNode = GnlFunctionOnSet (Function);
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node1), i);
        GnlInjectAllVarInNode (SonI, NewNode);
        BListElt (GnlNodeSons (Node1), i) = (int)(*NewNode);
    }

    *NewNode = Node1;
}

/*-----*/
/* GnlInjectAllVar */
/*-----*/
```

gnlinjct.c

```
GNL_STATUS GnlInjectAllVar (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR  Vari;
    GNL_FUNCTION FunctionI;
    GNL_NODE NewNode;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (Vari);
        GnlInjectAllVarInNode (GnlFunctionOnSet (FunctionI), &NewNode);
        SetGnlFunctionOnSet (FunctionI, NewNode);
    }

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        if (GnlVarDir (Vari) == GNL_VAR_LOCAL)
        {
            BListDelInsert (GnlFunctions (Gnl), i+1);
            i--;
            SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)-1);
        }
    }

    BSize (GnlLocals (Gnl)) = 0;

    return (GNL_OK);
}

/*-----
/* GnlGetDadsInGnlFromNode
*/
-----*/
GNL_STATUS GnlGetDadsInGnlFromNode (Node, Father)
    GNL_NODE Node;
    GNL_NODE Father;
{
    int          i;
    GNL_VAR  Var;
    GNL_NODE SonI;
    BLIST     NewList;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        if (GnlVarDads (Var) == NULL)
        {
            if (BListCreateWithSize (1, &NewList))
                return (GNL_MEMORY_FULL);
            SetGnlVarDads (Var, NewList);
        }
    }
}
```

```

    }
    if (BListAddElt (GnlVarDads (Var), (int)Node))
        return (GNL_MEMORY_FULL);

    if (Father == NULL)
        return (GNL_OK);

    if (GnlNodeDads (Node) == NULL)
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        SetGnlNodeDads (Node, NewList);
    }
    if (BListAddElt (GnlNodeDads (Node), (int)Father))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

if (GnlNodeDads (Node) == NULL)
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeDads (Node, NewList);
}
if (BListAddElt (GnlNodeDads (Node), (int)Father))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlGetDadsInGnlFromNode (SonI, Node))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlGetDadsInGnl
 */
GNL_STATUS GnlGetDadsInGnl (Gnl)
    GNL      Gnl;
{
    int          i;
    GNL_VAR  VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE NodeI;
    BLIST     NewList;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        NodeI = GnlFunctionOnSet (FunctionI);
        if (GnlNodeDads (NodeI) == NULL)

```

gnlinjct.c

```
{  
    if (BListCreateWithSize (1, &NewList))  
        return (GNL_MEMORY_FULL);  
    SetGnlNodeDads (NodeI, NewList);  
}  
if (BListAddElt (GnlNodeDads (NodeI), (int)VarI))  
    return (GNL_MEMORY_FULL);  
if (GnlGetDadsInGnlFromNode (NodeI, NULL))  
    return (GNL_MEMORY_FULL);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlInjectSingleVar */  
/*-----*/  
/* This procedure re-injects local Functions F such that NbOccur (F) is */  
/* equal to 1. */  
/*-----*/  
GNL_STATUS GnlInjectSingleVar (Gnl)  
    GNL      Gnl;  
{  
    int          i;  
    int          NbOccur;  
    GNL_FUNCTION FunctionI;  
    GNL_VAR     VarI;  
    float       MaxOccur;  
  
    GnlResetDadsInVar (Gnl);  
  
    /* we compute the number of Dads for Variable. */  
    GnlGetNumberDadsFromGnl (Gnl);  
  
    GnlInjectVarWithUniqueDad (Gnl);  
  
    GnlResetDadsInVar (Gnl);  
  
    /* we remove the unused variables. */  
    if (GnlRemoveUnusedVar (Gnl))  
        return (GNL_MEMORY_FULL);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlInjectSingleVarQuick */  
/*-----*/  
/* Quick version of 'GnlInjectSingleVar'. */  
/* This procedure re-injects local Functions F such that NbOccur (F) is */  
/* equal to 1. */  
/*-----*/  
GNL_STATUS GnlInjectSingleVarQuick (Gnl)  
    GNL      Gnl;  
}
```

gnlinjct.c

```
int          i;
int          NbOccur;
GNL_FUNCTION FunctionI;
GNL_VAR    Vari;
float       MaxOccur;

return (GNL_OK);
}

/*-----
/* GnlInjectSingleVarWithMaxLitt
*/
/*-----*/
/* This procedure re-injects local Functions F such that NbOccur (F) is */
/* equal to 1 and Number of litt. of F are less than MaxLitt           */
/*-----*/
GNL_STATUS GnlInjectSingleVarWithMaxLitt (Gnl, MaxLitt)
    GNL      Gnl;
    int     MaxLitt;
{
    int          i;
    int          NbOccur;
    GNL_FUNCTION FunctionI;
    GNL_VAR    Vari;
    float       MaxOccur;

    GnlResetDadsInVar (Gnl);

    /* we compute the number of Dads for Variable.                      */
    GnlGetNumberDadsFromGnl (Gnl);

    GnlInjectVarWithUniqueDadWithMaxLitt (Gnl, MaxLitt);

    GnlResetDadsInVar (Gnl);

    /* we remove the unused variables.                                     */
    if (GnlRemoveUnusedVar (Gnl))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlInject
*/
/*-----*/
GNL_STATUS GnlInjectWithMaxDepth (Gnl, Limit)
    GNL      Gnl;
    int     Limit;
{
    int          i;
    int          NbLitt;
    int          NbOccur;
    GNL_FUNCTION FunctionI;
    GNL_VAR    Vari;
```

gnlinjct.c

```
float      MaxOccur;
int       MaxLevel;
int       Level;

GnlSortFunctionsOnInvertLevelStoreLevel (Gnl);
VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), 0);
MaxLevel = (int)GnlVarHook (VarI);
printf ("Maxlevel = %d\n", MaxLevel);

GnlResetDadsInVar (Gnl);

/* we compute the number of Dads for Variable. */  

GnlGetNumberDadsFromGnl (Gnl);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    if (GnlVarDir (VarI) != GNL_VAR_LOCAL)
        continue;

    /* we substitute only LOCAL variables */  

    Level = (int)GnlVarHook (VarI);

    NbOccur = (int)GnlVarDads (VarI);

    if (Level >= MaxLevel-2)
    {
        /* We replace all occurences of 'VarI' by its expression */
        /* Node. This variable 'VarI' is then removed from the Gnl. */
        GnlInjectLocalVar (Gnl, FunctionI, &NbOccur);

        GnlRemoveVarFromGnlLocals (Gnl, VarI);
        BlistDelShift (GnlFunctions (Gnl), i+1);
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)-1);

        i--;
    }
}

GnlResetDadsInVar (Gnl);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    SetGnlVarHook (VarI, NULL);
}

return (GNL_OK);
}

/*----- EOF -----*/
```

gnllinklib.c

```
/*
 *-----*/
/*      File:          gnllinklib.c
 *      Version:        1.1
 *      Modifications: -
 *      Documentation: -
 */
/*      Description:          */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlestim.h"
#include "gnlmap.h"

#include "blist.e"

/*
 *-----*/
/* EXTERN          */
/*-----*/
/*
 *-----*/
/* GnlCreateLinkLibraryHashTable          */
/*-----*/
/* This procedure scans the LIBC 'Gnllibc' and stores each cell in a      */
/* Hash table. Each cell is stored according to its name.          */
/*-----*/
#define LINK_LIB_HASH_TABLE_SIZE    100
GNL_STATUS GnlCreateLinkLibraryHashTable (Gnllibc, LibHashTable)
    LIBC_LIB    Gnllibc;
    BLIST      *LibHashTable;
{
    int             i;
    LIBC_CELL      Celli;
    unsigned int    Key;
    BLIST          NewList;

    if (BListCreateWithSize (LINK_LIB_HASH_TABLE_SIZE, LibHashTable))
        return (GNL_MEMORY_FULL);

    for (i=0; i<LINK_LIB_HASH_TABLE_SIZE; i++)

```

```

{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (*LibHashTable, (int)NewList))
        return (GNL_MEMORY_FULL);
}

CellI = LibCells (GnlLibc);
for (; CellI != NULL; CellI = LibCellNext (CellI))
{
    Key = KeyOfName (LibCellName (CellI), LINK_LIB_HASH_TABLE_SIZE);
    NewList = (BLIST)BListElt (*LibHashTable, Key);
    if (BListAddElt (NewList, (int)CellI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----*/
/* GnlLinkLibRec */
/*-----*/
GNL_STATUS GnlLinkLibRec (Nw, Gnl, LibHashTable)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    BLIST           LibHashTable;
{
    GNL             TopGnl;
    BLIST           Components;
    GNL_COMPONENT   ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    int              i;
    int              j;
    GNL             GnlCompoI;
    unsigned int     Key;
    BLIST           NewList;
    LIBC_CELL       CellJ;
    BLIST           Interface;

/* Already traversed and considered this 'Gnl'. */
if (GnlTag (Gnl) == GnlNetworkTag (Nw))
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

Components = GnlComponents (Gnl);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);
}

```

```

/* If the component 'UserCompoI' has no Gnl definition. */
if (!GnlCompoI)
{
    /* Right now no lib cell definition for 'UserCompoI'. */

    /* No definition so it can be a direct library cell */
    Key = KeyOfName (GnlUserComponentName (UserCompoI),
                      LINK_LIB_HASH_TABLE_SIZE);
    NewList = (BLIST)BListElt (LibHashTable, Key);
    for (j=0; j<BListSize (NewList); j++)
    {
        CellJ = (LIBC_CELL)BListElt (NewList, j);
        if (!strcmp (GnlUserComponentName (UserCompoI),
                     LibCellName (CellJ)))
        {
            /* We link the component with its corresponding
             * library cell. */
            SetGnlUserComponentCellDef (UserCompoI, CellJ);
            break;
        }
    }
    if (!GnlUserComponentCellDef (UserCompoI))
    {
        fprintf (stderr,
 "# WARNING: component <%s> has no corresponding library cell\n",
                  GnlUserComponentName (UserCompoI));
    }
    continue;
}

/* It is a user box and we scan recursively. */
if (GnlLinkLibRec (Nw, GnlCompoI,
                    LibHashTable))
    return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlLinkLib
 */
/*
 * This procedure analyzes recursively the netlist 'Gnl' and makes the
 * link between each GNL_USER_COMPONENT and their corresponding cell in
 * the LIBC library. From a GNL_USER_COMPONENT 'compo' this cell is
 * accessible thru field 'GnlUserComponentCellDef (compo)'.
 */
GNL_STATUS GnlLinkLib (Nw, Gnl, GnlLibc)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    LIBC_LIB        GnlLibc;
{
    BLIST      LibHashTable;
    int         i;
    BLIST      NewList;

```

gnllinklib.c

```
if (GnlCreateLinkLibraryHashTable (GnlLibc, &LibHashTable))
    return (GNL_MEMORY_FULL);

SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

if (GnlLinkLibRec (Nw, Gnl, LibHashTable))
    return (GNL_MEMORY_FULL);

/* Freeing the 'LibHashTable' hash table. */  

for (i=0; i<BListSize (LibHashTable); i++)
{
    NewList = (BLIST)BListElt (LibHashTable, i);
    BListQuickDelete (&NewList);
}
BListQuickDelete (&LibHashTable);

return (GNL_OK);
}

/*-----*/
```

gnlmain.c

```
/*-----*/
/*      File:          gnlmain.c
/*      Modifications: -
/*      Documentation: -
/*      Class: none
/*      Inheritance:
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "gnlopt.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnloption.h"

/*-----*/
/* GLOBAL VARIABLE */
/*-----*/
extern GNL_ENV      G_GnlEnv;

/*-----*/
/* GnlPrintInvite */
/*-----*/
void GnlPrintInvite ()
{
    int             i;

    fprintf (stderr, "#-----\n");
    fprintf (stderr, "#\n");
    fprintf (stderr,
            "# %s version %s - RTL Prediction & Synthesis      ",
            GNL_TOOL_NAME,
            GNL_MAPIT_VERSION);
    for (i=0; i<strlen (GNL_MAPIT_VERSION); i++)
        fprintf (stderr, " ");
    fprintf (stderr, "#\n");
    fprintf (stderr, "#\n");
    fprintf (stderr, "#\n");
    fprintf (stderr, "# Copyright (c) Avant! Corporation 1994-99. All rights
reserved.          #\n");
    fprintf (stderr, "#\n");
    fprintf (stderr, "#-----\n");
}

}
```

```

/*
 *-----*
 * main
 *-----*
 int main (argc, argv)
    int      argc;
    char    **argv;
{
    GnlPrintInvite ();

    /* This call creates and set global variable 'G_GnlEnv'. Options      */
    /* will be stored in this object.                                         */
    if (GnlCreateGnlEnv ())
        exit (1);

    /* we parse and analyze the options and store them in global env.      */
    /* variable 'G_GnlEnv'.                                                 */
    GnlParseCommandLine (G_GnlEnv, argc, argv);

    /* We check if the required options are set.                            */
    GnlCheckOptions (G_GnlEnv);

    /* we are in the Verification mode.                                     */
    switch (GnlEnvMode ()) {
        case GNL_MODE_VERIFICATION:
            if (GnlEcVerification ())
                exit (1);
            exit (0);

        case GNL_MODE_TRANSLATE:
            if (GnlTranslate ())
                exit (1);
            exit (0);

        case GNL_MODE_SYNTHESIS:
            if (GnlSynthesize ())
                exit (1);
            exit (0);

        case GNL_MODE_ESTIMATION:
            if (GnlEstimate ())
                exit (1);
            exit (0);

        case GNL_MODE_POST_OPTIMIZATION:
            if (GnlPostOptimize ())
                exit (1);
            exit (0);
    }
}
/*
 *-----*

```

gnlmint.c

```
/*-----*/
/*
 *      File:          gnlmint.c
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description:
 */
/*-----*/
#include <stdio.h>

#include "gnlmint.h"

/* ----- */
/* MintInitWithValue */
/* ----- */
void MintInitWithValue (cells, nb_cells, value)
    unsigned   *cells;
    int        nb_cells;
    unsigned   value;
{
    while (nb_cells--)
        *(cells++) = value;
}

/* ----- */
/* MintCreate */
/* ----- */
MINT_STATUS MintCreate (nb_cells, cells)
    int        nb_cells;
    unsigned **cells;
{
    if ((*cells = (unsigned *) calloc ((unsigned) (nb_cells + 1),
                                      (unsigned) sizeof (unsigned))) == NULL)
        return (MINT_MEMORY_FULL);

    return (MINT_OK);
}

/* ----- */
/* MintCreateInitValue */
/* ----- */
MINT_STATUS MintCreateInitValue (nb_cells, value, cells)
    int        nb_cells;
    unsigned   value;
    unsigned **cells;
{
    MINT_STATUS      MintStatus;

    if ((MintStatus = MintCreate (nb_cells, cells)))
        return (MintStatus);

    MintInitWithValue ((*cells), nb_cells, value);
```

gnlmint.c

```
    return (MINT_OK);
}

/*
 * MintCreateInit0
 */
MINT_STATUS MintCreateInit0 (nb_cells, cells)
    int          nb_cells;
    unsigned **cells;
{
    return (MintCreateInitValue (nb_cells, 0, cells));
}

/*
 * MintCardinalOneCell
 */
int MintCardinalOneCell (cell)
    unsigned cell;
{
    int card;

    card = 0;
    while (cell)
    {
        card++;
        cell &= (cell - 1);
    };

    return (card);
}

/*
 * MintCardinalAllCells
 */
int MintCardinalAllCells (cells, nb_cells)
    unsigned *cells;
    int          nb_cells;
{
    int      card;

    card = 0;
    cells += nb_cells;

    while (nb_cells--)
        card += MintCardinalOneCell (*(--cells));

    return (card);
}

/*
 * MintCopy
 */
MINT_STATUS MintCopy (cells, nb_cells, copy_cells)
    unsigned *cells;
    int          nb_cells;
    unsigned **copy_cells;
{
```

gnlmint.c

```
if (MintCreate (nb_cells, copy_cells))
    return (MINT_MEMORY_FULL);

if ((*copy_cells) != NULL)
    while (nb_cells--)
        (*copy_cells) [nb_cells] = cells[nb_cells];

return (MINT_OK);
}

/*
/* MintBitValue
*/
int MintBitValue (mint, index)
    unsigned *mint;
    int         index;

{
    int cell_number;
    int rank_in_cell;

locate (index, BITS_PER_INT, cell_number, rank_in_cell);
return ((mint[cell_number] & (1 << (rank_in_cell - 1))) != 0);
}

/*
/* MintLocateFirstOne
*/
int MintLocateFirstOne (Mint, NbCells, CellNb, VarMask)
    unsigned *Mint;
    int         NbCells;
    int         *CellNb;
    unsigned *VarMask;
{

    while (NbCells-- && !Mint[NbCells]);

    if ((NbCells >= 0) && (*VarMask = Mint[NbCells]))
    {
        *CellNb = NbCells;
        *VarMask &= ~(*VarMask - 1);
        return (1);
    }

    return (0);
}

/*
/* MintNotNull
*/
int MintNotNull (cells, nb_cells)
    unsigned *cells;
    int         nb_cells;

{
```

gnlmint.c

```
while (nb_cells--)
    if (*cells++)
        return (1);
    return (0);
}

/* -----
/* MintIdentical                               */
/* ----- */
int MintIdentical (cells1, cells2, nb_cells)
    unsigned *cells1;
    unsigned *cells2;
    int      nb_cells;
{
    while (nb_cells--)
        if (cells1[nb_cells] != cells2[nb_cells])
            return (0);
    return (1);
}

/* -----
/* MintAddOneCell                            */
/* ----- */
unsigned MintAddOneCell (cell, added_value)
    unsigned *cell;
    unsigned added_value;

{
    if (*cell <= ~0 - added_value)
    {
        *cell += added_value;
        return (0);
    }
    (*cell) -= ~0 - added_value;
    (*cell)--;
    return (1);
}

/* -----
/* MintIncrement                             */
/* ----- */
unsigned MintIncrement (cells, nb_cells)
    unsigned *cells;
    int      nb_cells;
{
    unsigned carry;

    do
        carry = MintAddOneCell (cells++, 1);
    while (carry && (--nb_cells));

    return (carry);
}
```

gnlmint.c

```
/* ----- */
/* MintSetBitValue */          */
/* ----- */
/* Set the index-th bit of mint to 1. */      */
/* ----- */
void MintSetBitValue (mint, index)
    unsigned *mint;
    int index;
{
    int cell_number;
    int rank_in_cell;

    locate (index, BITS_PER_INT, cell_number, rank_in_cell);
    mint[cell_number] |= (1 << (rank_in_cell - 1));
}

/* ----- */
/* MintResetBitValue */          */
/* ----- */
/* Set the index-th bit of mint to 0. */      */
/* ----- */
void MintResetBitValue (mint, index)
    unsigned *mint;
    int index;
{
    int cell_number;
    int rank_in_cell;

    locate (index, BITS_PER_INT, cell_number, rank_in_cell);
    mint[cell_number] &= ~(1 << (rank_in_cell - 1));
}

/* ----- */
/* MintNOT */                      */
/* ----- */
MINT_STATUS MintNOT (op1, nb_cells, result)
    unsigned *op1;
    int nb_cells;
    unsigned **result;
{
    unsigned *res;

    if (MintCreate (nb_cells, &res))
        return (MINT_MEMORY_FULL);

    *result = res;
    while (nb_cells--)
        *(res++) = ~*(op1++);

    return (MINT_OK);
}

/* ----- */
```

gnlmint.c

```
/* MintAND */  
/* ----- */  
MINT_STATUS MintAND (op1, op2, nb_cells, result)  
    unsigned *op1;  
    unsigned *op2;  
    int nb_cells;  
    unsigned **result;  
{  
    unsigned *res;  
  
    if (MintCreate (nb_cells, &res))  
        return (MINT_MEMORY_FULL);  
  
    *result = res;  
    while (nb_cells--)  
        *(res++) = *(op1++) & *(op2++);  
  
    return (MINT_OK);  
}  
  
/* ----- */  
/* MintXNOR */  
/* ----- */  
MINT_STATUS MintXNOR (op1, op2, nb_cells, result)  
    unsigned *op1;  
    unsigned *op2;  
    int nb_cells;  
    unsigned **result;  
{  
    unsigned *res;  
  
    if (MintCreate (nb_cells, &res))  
        return (MINT_MEMORY_FULL);  
  
    *result = res;  
    while (nb_cells--)  
        *(res++) = ~(*(op1++) ^ *(op2++));  
  
    return (MINT_OK);  
}  
  
/* ----- */  
/* MintNOTNoCreate */  
/* ----- */  
int MintNOTNoCreate (op1, res, nb_cells)  
    unsigned *op1;  
    unsigned *res;  
    int nb_cells;  
{  
    int exists;  
  
    exists = 0;  
    while (nb_cells--)
```

gnlmint.c

```
if (*res++ = ~(*op1++))
exists = 1;

return (exists);
}

/*
/* MintANDNoCreate
*/
int MintANDNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int      nb_cells;
{
    int      exists;

    exists = 0;
    while (nb_cells--)
        if (*res++ = *(op1++) & *(op2++))
            exists = 1;

    return (exists);
}

/*
/* MintORNoCreate
*/
int MintORNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int      nb_cells;
{
    int      exists;

    exists = 0;
    while (nb_cells--)
        if (*res++ = *(op1++) | *(op2++))
            exists = 1;

    return (exists);
}

/*
/* MintNORNNoCreate
*/
int MintNORNNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int      nb_cells;
{
    int      exists;
```

gnlmint.c

```
exists = 0;
while (nb_cells--)
    if (*res++ = -(*op1++) | *op2++))
        exists = 1;

    return (exists);
}

/*
/* MintXORNoCreate
*/
int MintXORNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int nb_cells;
{
    while (nb_cells--)
        *(res++) = *(op1++) ^ *(op2++);

}

/*
/* MintXNORNoCreate
*/
int MintXNORNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int nb_cells;
{
    while (nb_cells--)
        *(res++) = ~(*op1++) ^ *(op2++);

}

/*
/* MintIncluded
*/
int MintIncluded (cells1, cells2, nb_cells)
    unsigned *cells1;
    unsigned *cells2;
    int nb_cells;
{
    while (nb_cells--)
        if (*cells1++ & ~*(cells2++))
            return (0);

    return (1);
}

/*
/* MintOR
*/
```

gnlmint.c

```
/* ----- */
MINT_STATUS MintOR (op1, op2, nb_cells, result)
    unsigned *op1;
    unsigned *op2;
    int nb_cells;
    unsigned **result;
{
    unsigned *res;

    if (MintCreate (nb_cells, &res))
        return (MINT_MEMORY_FULL);

    *result = res;

    while (nb_cells--)
        *(res++) = *(op1++) | *(op2++);

    return (MINT_OK);
}

/* ----- */
/* MintNAND */
/* ----- */
MINT_STATUS MintNAND (op1, op2, nb_cells, result)
    unsigned *op1;
    unsigned *op2;
    int nb_cells;
    unsigned **result;
{
    unsigned *res;

    if (MintCreate (nb_cells, &res))
        return (MINT_MEMORY_FULL);

    *result = res;
    while (nb_cells--)
        *(res++) = ~(*(op1++) & *(op2++));

    return (MINT_OK);
}

/* ----- */
/* MintNANDNoCreate */
/* ----- */
int MintNANDNoCreate (op1, op2, res, nb_cells)
    unsigned *op1;
    unsigned *op2;
    unsigned *res;
    int nb_cells;
{
    int exists;

    exists = 0;
```

gnlmint.c

```
while (nb_cells--)
    if (*res++ = ~(*op1++ & *op2++))
        exists = 1;
return (exists);
}

/*
 * MintNOR
 */
MINT_STATUS MintNOR (op1, op2, nb_cells, result)
    unsigned *op1;
    unsigned *op2;
    int nb_cells;
    unsigned **result;
{
    unsigned *res;

    if (MintCreate (nb_cells, &res))
        return (MINT_MEMORY_FULL);

    *result = res;

    while (nb_cells--)
        *(res++) = ~(*op1++ | *op2++);

    return (MINT_OK);
}

/*
 * MintXOR
 */
MINT_STATUS MintXOR (op1, op2, nb_cells, result)
    unsigned *op1;
    unsigned *op2;
    int nb_cells;
    unsigned **result;
{
    unsigned *res;

    if (MintCreate (nb_cells, &res))
        return (MINT_MEMORY_FULL);

    *result = res;
    while (nb_cells--)
        *(res++) = *(op1++) ^ *(op2++);

    return (MINT_OK);
}

/*
 */
```

gnlmint.h

```
/*----- */  
/*  
 *      File:          gnlmint.h  
 *      Version:        1.1  
 *      Modifications: -  
 *      Documentation: -  
 *  
 *      Description:  
 *----- */  
#ifndef GNLMINT_H  
#define GNLMINT_H  
  
/*----- */  
/* MINT_STATUS */  
/*----- */  
typedef enum {  
    MINT_OK,  
    MINT_MEMORY_FULL  
} MINT_STATUS;  
  
/* ----- */  
/* Macros */  
/* ----- */  
#define BITS_PER_INT (8*sizeof(int))  
  
#define NbOfCells(NbVar,CellSize) \  
    ((NbVar)?(1+((int)((NbVar)-1)/(CellSize))):1)  
  
#define locate(VarIndex,CellSize,CellNumber,RankInCell) \  
    (RankInCell=(int)(VarIndex)-(CellNumber=(int)((VarIndex)-  
1)/(CellSize)))*(CellSize))  
  
#define IntDiff(x,y) ((x) > (y) ? (x) - (y) : (y) - (x))  
#define MintHasMoreOrLessThanOne_1(I) \  
    (((I) == 0 ? -1 : ((I) & ((I) - 1) ? 1 : 0))  
/* ----- EOF ----- */  
#endif
```

gnlopt.c

```
/*-----*/
/*
 *      File:          gnlopt.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 *
 *      Description:          */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnlopt.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnloption.h"

#include "blist.e"

/*-----*/
/* EXTERN          */
/*-----*/
extern GNL_STATUS      LsDivGainFunction ();
extern GNL_STATUS      LsDivChoiceFunction ();
extern GNL_STATUS      LsExtractGainFunction ();
extern GNL_STATUS      LsExtractChoiceFunction ();
extern GNL_STATUS      LsDivGainFunctionEqGates ();
extern GNL_STATUS      LsExtractGainFunctionEqGates ();
extern double          GnlGetNetworkGateArea ();
extern GNL_ENV          G_GnlEnv;

/*-----*/
/* Global variables.          */
/*-----*/
LS_OPT_PANEL G_OptPanel;

/*-----*/
/* GnlOptCreatePanel          */
/*-----*/
GNL_STATUS GnlOptCreatePanel (NewOptPanel)
    LS_OPT_PANEL    *NewOptPanel;
{
    if (((*NewOptPanel) = (LS_OPT_PANEL)
        calloc (1, sizeof (LS_OPT_PANEL_REC))) == NULL)
        return (GNL_MEMORY_FULL);
```

gnlopt.c

```
    return (GNL_OK);
}

/*-----
/* GnlOptSetDefaultPanel
*/
void GnlOptSetDefaultPanel (OptPanel)
    LS_OPT_PANEL    OptPanel;
{
    float      CombGates;
    float      SeqGates;

    SetLsOptPanelDoOpt (OptPanel, 1);
    SetLsOptPanelDoDiv (OptPanel, 1);
    SetLsOptPanelDoExt (OptPanel, 1);

    SetLsOptPanelMaxDivKer (OptPanel, 5000);
    SetLsOptPanelMaxExtKer (OptPanel, 1000);

    SetLsOptPanelDivGainFunc (OptPanel, LsDivGainFunction);
    SetLsOptPanelDivChoiceFunc (OptPanel, LsDivChoiceFunction);

    SetLsOptPanelExtGainFunc (OptPanel, LsExtractGainFunction);
    SetLsOptPanelExtChoiceFunc (OptPanel, LsExtractChoiceFunction);

    SetLsOptMaxLitToInject (OptPanel, 1);

    SetLsOptMaxFlattenCubes (OptPanel, 200);
}

/*-----
/* GnlCreateSynthesisPanel
*/
GNL_STATUS GnlCreateSynthesisPanel (Force, Criter, Inject, Panel)
    GNL_OPT_FORCE  Force;
    GNL_CRITERION Criter;
    int           Inject;
    LS_OPT_PANEL   *Panel;
{
    /* Setting the Optimization panel values
    if (GnlOptCreatePanel (Panel))
        return (GNL_MEMORY_FULL);

    GnlOptSetDefaultPanel ((*Panel));

    if (Force == GNL_LOW_OPT)
    {
        /* We inhibit the optimisation phase.
        SetLsOptPanelDoOpt ((*Panel), 0);
        SetLsOptPanelMapEffort ((*Panel), 0);
    }
    else if (Force == GNL_MED_OPT)
    {
        SetLsOptMaxFlattenCubes ((*Panel), 200);
    }
}
```

gnlopt.c

```
        SetLsOptPanelMapEffort ((*Panel), 0);
    }
else
{
    SetLsOptPanelDoOpt ((*Panel), 2);
    SetLsOptMaxFlattenCubes ((*Panel), 400);
    SetLsOptPanelMapEffort ((*Panel), 1);
}

if (Criter == GNL_AREA)
    SetLsOptPanelCriter ((*Panel), 1); /* Area */
else
    SetLsOptPanelCriter ((*Panel), 0); /* Timing */

SetLsOptMaxInject ((*Panel), Inject);

return (GNL_OK);
}

/*-----
/* GnlPerformAutomaticInject */
/*-----
/* we perform an automatic injection of the Gnl. The injection takes */
/* account the features of the Gnl (number litterals, ...). This */
/* must be studied in order to improve the performances. */
/*-----*/
GNL_STATUS GnlPerformAutomaticInject (Gnl, OptPanel)
    GNL          Gnl;
    LS_OPT_PANEL OptPanel;
{
    GNL_INFO GnlInfo;

    /* We do not enter for now in this branch because we need to improve */
    /* the automatic flattening. */
    if (GnlInject (Gnl, LsOptMaxLitToInject (OptPanel)))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);

    if (GnlGetGnlInfo (Gnl, &GnlInfo))
        return (GNL_MEMORY_FULL);

    if (GnlInfoNbLit (GnlInfo) < 500)
    {
        if (GnlInject (Gnl, 100))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (GnlInject (Gnl, LsOptMaxLitToInject (OptPanel)))
            return (GNL_MEMORY_FULL);
    }

    free (GnlInfo);
    return (GNL_OK);
}
```

gnlopt.c

```
-----*/
/* GnlLiteralCompare */
-----*/
/* Comparison function to compare the number of literals between 2 Gnls */
-----*/
int GnlLiteralCompare (Gnl1, Gnl2)
    GNL          *Gnl1;
    GNL          *Gnl2;
{
    int           NbLit1;
    int           NbLit2;

    NbLit1 = GnlNbLitt (*Gnl1);
    NbLit2 = GnlNbLitt (*Gnl2);

    if (NbLit1 < NbLit2)
        return (1);

    if (NbLit1 == NbLit2)
        return (0);

    return (-1);
}

-----*/
/* GnlIsoListCmp */
-----*/
/* Comparison function to compare the complexity of two Iso lists */
-----*/
int GnlIsoListCmp (IsoList1, IsoList2)
    BLIST         *IsoList1;
    BLIST         *IsoList2;
{
    int           NbLit1;
    int           NbLit2;
    int           NbGnl1;
    int           NbGnl2;
    int           Complex1;
    int           Complex2;

    NbLit1 = GnlNbLitt ((GNL)BListElt (*IsoList1, 0));
    NbLit2 = GnlNbLitt ((GNL)BListElt (*IsoList2, 0));

    NbGnl1 = (BListSize (*IsoList1) == 1 ? 1 : BListSize (*IsoList1)-3);
    NbGnl2 = (BListSize (*IsoList2) == 1 ? 1 : BListSize (*IsoList2)-3);

    Complex1 = NbLit1*NbGnl1;
    Complex2 = NbLit2*NbGnl2;

    if (Complex1 < Complex2)
        return (1);

    if (Complex1 > Complex2)
        return (-1);
```

gnlopt.c

```
    return (0);
}

/*-----*/
/* GnlSortListOfGnl */
/*-----*/
/* This procedure computes the number of literals of each Gnl and store */
/* the information in the Gnl structure. Then it sorts the list of Gnls */
/* according to the number of literals of each Gnl. */
/*-----*/
void GnlSortListOfGnl (ListGnls)
    BLIST      ListGnls;
{
    int             i;
    GNL            GnlI;
    int             NbLit;

    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnlI = (GNL)BListElt (ListGnls, i);
        NbLit = GnlGetGnlNbLiterals (GnlI);
        SetGnlNbLitt (GnlI, NbLit);
    }

    qsort (BListAddress (ListGnls), BListSize (ListGnls), sizeof (GNL),
           GnlLiteralCompare);
}

/*-----*/
/* GnlSortListOfHashListIsoGnls */
/*-----*/
/* we sort the iso gnls sub-lists in the decreasing order of the */
/* NbLit (RefGnl)*BListSize (sub-list). */
/*-----*/
void GnlSortListOfHashListIsoGnls (HashListIsoGnls)
    BLIST      HashListIsoGnls;
{
    qsort (BListAddress (HashListIsoGnls), BListSize (HashListIsoGnls),
           sizeof (BLIST), GnlIsoListCmp);
}

/*-----*/
/* GnlIsomorphNodes */
/*-----*/
int GnlIsomorphNodes (Node1, Node2)
    GNL_NODE Node1;
    GNL_NODE Node2;
{
    int             i;
    GNL_VAR        Var1;
    GNL_VAR        Var2;
    BLIST         Sons1;
```

gnlopt.c

```
BLIST          Sons2;
GNL_NODE       Node1I;
GNL_NODE       Node2I;

if (GnlNodeOp (Node1) != GnlNodeOp (Node2))
    return (0);

if (GnlNodeOp (Node1) == GNL_CONSTANTE)
    return ((GnlNodeSons (Node1) == GnlNodeSons (Node2)));

if (GnlNodeOp (Node1) == GNL_VARIABLE)
{
    Var1 = (GNL_VAR)GnlNodeSons (Node1);
    Var2 = (GNL_VAR)GnlNodeSons (Node2);

    if (GnlVarHook (Var1))
    {
        if (GnlVarHook (Var1) != (void*)Var2)
            return (0);
        return (1);
    }

    if (GnlVarIsVss (Var1))
    {
        if (GnlVarIsVss (Var2))
            return (1);
        return (0);
    }

    if (GnlVarIsVdd (Var1))
    {
        if (GnlVarIsVdd (Var2))
            return (1);
        return (0);
    }

    SetGnlVarHook (Var1, Var2);
    return (1);
}

Sons1 = GnlNodeSons (Node1);
Sons2 = GnlNodeSons (Node2);
if (BListSize (Sons1) != BListSize (Sons2))
    return (0);

for (i=0; i<BListSize (GnlNodeSons (Node1)); i++)
{
    Node1I = (GNL_NODE)BListElt (Sons1, i);
    Node2I = (GNL_NODE)BListElt (Sons2, i);
    if (!GnlIsomorphNodes (Node1I, Node2I))
        return (0);
}

return (1);
}
```

```

/*-----*/
/* GnlAreIsomorphic */
/*-----*/
/* This procedure returns 1 if the two Gnls are strictly isomorph and */
/* 0 otherwise. */
/*-----*/
int GnlAreIsomorphic (Gnl1, Gnl2)
    GNL          Gnl1;
    GNL          Gnl2;
{
    int           i;
    GNL_VAR      VarI;
    GNL_VAR      Out1;
    GNL_VAR      Out2;
    GNL_FUNCTION Function1;
    GNL_FUNCTION Function2;
    GNL_NODE     Node1;
    GNL_NODE     Node2;

    if (BListSize (GnlInputs (Gnl1)) != BListSize (GnlInputs (Gnl2)))
        return (0);

    if (BListSize (GnlOutputs (Gnl1)) != BListSize (GnlOutputs (Gnl2)))
        return (0);

    if (BListSize (GnlLocals (Gnl1)) != BListSize (GnlLocals (Gnl2)))
        return (0);

    /* We reset the link var (using Hook field) for inputs/outputs/locals*/
    /* before verifying the isomorphism. */
    for (i=0; i<BListSize (GnlInputs (Gnl1)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlInputs (Gnl1), i);
        SetGnlVarHook (VarI, NULL);
    }
    for (i=0; i<BListSize (GnlOutputs (Gnl1)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlOutputs (Gnl1), i);
        SetGnlVarHook (VarI, NULL);
    }
    for (i=0; i<BListSize (GnlLocals (Gnl1)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlLocals (Gnl1), i);
        SetGnlVarHook (VarI, NULL);
    }

    for (i=0; i<BListSize (GnlOutputs (Gnl1)); i++)
    {
        Out1 = (GNL_VAR)BListElt (GnlOutputs (Gnl1), i);
        Out2 = (GNL_VAR)BListElt (GnlOutputs (Gnl2), i);
        Function1 = GnlVarFunction (Out1);
        Function2 = GnlVarFunction (Out2);
        Node1 = GnlFunctionOnSet (Function1);
        Node2 = GnlFunctionOnSet (Function2);
    }
}

```

gnlopt.c

```
/* The Output has been already linked so we verify that it is      */
/* linked to the outvar 'Out2'.                                         */
if (GnlVarHook (Out1))
{
    if (GnlVarHook (Out1) != (void*)Out2)
        return (0);
}

SetGnlVarHook (Out1, Out2);

if (!GnlIsomorphNodes (Node1, Node2))
    return (0);
}

for (i=0; i<BListSize (GnlLocals (Gnl1)); i++)
{
    Out1 = (GNL_VAR)BListElt (GnlLocals (Gnl1), i);
    Out2 = (GNL_VAR)BListElt (GnlLocals (Gnl2), i);
    Function1 = GnlVarFunction (Out1);
    Function2 = GnlVarFunction (Out2);
    Node1 = GnlFunctionOnSet (Function1);
    Node2 = GnlFunctionOnSet (Function2);

    /* The Output has been already linked so we verify that it is      */
    /* linked to the outvar 'Out2'.                                         */
    if (GnlVarHook (Out1))
    {
        if (GnlVarHook (Out1) != (void*)Out2)
            return (0);
    }

    SetGnlVarHook (Out1, Out2);

    if (!GnlIsomorphNodes (Node1, Node2))
        return (0);
    }

    return (1);
}

/*-----
/* GnlModifyIsomorphicGnl                                         */
/*-----*/
/* This procedure modifies the interface of the GnlJ by sorting the   */
/* inputs and locals variables in order to have a direct bijection   */
/* between these variables and the one of the referenced Gnl 'GnlI'.   */
/*-----*/
GNL_STATUS GnlModifyIsomorphicGnl (GnlI, GnlJ)
    GNL          GnlI;
    GNL          GnlJ;
{
    int           i;
    GNL_VAR  VarI;
    BLIST       NodesSegments;

    GnlFreeNodesSegments (GnlJ);
```

gnlopt.c

```
if (BListCreate (&NodesSegments))
    return (GNL_MEMORY_FULL);

SetGnlNodesSegments (GnlJ, NodesSegments);
SetGnlFirstNode (GnlJ, NULL);
SetGnlLastNode (GnlJ, NULL);

/* We make a side effect on the inputs ordering of GnlJ in order to */
/* match the one to one correspondance with the inputs of 'GnLI'. */
for (i=0; i<BListSize (GnlInputs (GnLI)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlInputs (GnLI), i);
    BListElt (GnlInputs (GnlJ), i) = (int)GnlVarHook (VarI);
}

/* Same thing for the locals variables. */
for (i=0; i<BListSize (GnlLocals (GnLI)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlLocals (GnLI), i);
    BListElt (GnlLocals (GnlJ), i) = (int)GnlVarHook (VarI);
}

return (GNL_OK);
}

/*
 * GnlHashListIsomorphGnl
 */
/*
 * This procedure analyzes all the Gnls in the list 'ListGnls' and
 * group the ones which have the same structure (e.g isomorphe).
 * 'HashListIsoGnls' is a list of isomorphic Gnls. The first element of
 * the isomorphic list of Gnls is the Referenced Gnl. The second the
 * list of inputs of the Referenced Gnl duplicated. The third element
 * the list of locals of the Referenced Gnl duplicated. The following
 * elements are the isomorphic Gnls in which we have the removed the
 * Gnl nodes.
 */
GNL_STATUS GnlHashListIsomorphGnl (ListGnls, HashListIsoGnls)
BLIST ListGnls;
BLIST *HashListIsoGnls;
{
    int i;
    int j;
    GNL GnLI;
    GNL GnlJ;
    BLIST IsoList;
    BLIST NewList;

    if (GnlEnvFoldIsoPart ())
        fprintf (stderr, " Folding Identical Glue Logic Partitions...\n");

    if (BListCreate (HashListIsoGnls))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListGnls); i++)
```

```

{
    if (BListCreateWithSize (1, &IsoList))
        return (GNL_MEMORY_FULL);

    GnlI = (GNL)BListElt (ListGnls, i);

    if (BListAddElt (IsoList, (int)GnlI))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (*HashListIsoGnls, (int)IsoList))
        return (GNL_MEMORY_FULL);

    /* The user did not request any logic folding. */
    if (!GnlEnvFoldIsoPart ())
        continue;

    for (j=i+1; j<BListSize (ListGnls); j++)
    {
        GnlJ = (GNL)BListElt (ListGnls, j);

        /* 'GnlJ' has not the same number of literals so we can      */
        /* already stop.                                              */
        if (GnlNbLitt (GnlJ) != GnlNbLitt (GnlI))
            break;

        if (GnlAreIsomorphic (GnlI, GnlJ))
        {
            if (GnlModifyIsomorphicGnl (GnlI, GnlJ))
                return (GNL_MEMORY_FULL);

            /* The second element in the 'IsoList' having isomor. */
            /* Gnls is the original list of inputs. The third the */
            /* original list of Locals.                           */
            if (BListSize (IsoList) == 1)
            {
                /* We store the Inputs list of the referenced Gnl */
                if (BListCopyNoEltCr (GnlInputs (GnlI), &NewList))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (IsoList, (int)NewList))
                    return (GNL_MEMORY_FULL);
                /* We store the Locals list of the referenced Gnl */
                if (BListCopyNoEltCr (GnlLocals (GnlI), &NewList))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (IsoList, (int)NewList))
                    return (GNL_MEMORY_FULL);
                /* We store the Outputs list of the referenced Gnl */
                if (BListCopyNoEltCr (GnlOutputs (GnlI), &NewList))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (IsoList, (int)NewList))
                    return (GNL_MEMORY_FULL);
            }

            if (BListAddElt (IsoList, (int)GnlJ))
                return (GNL_MEMORY_FULL);
            BListDelShift (ListGnls, j+1);
            j--;
        }
    }
}

```

gnlopt.c

```
    }

    if (!GnlEnvFoldIsoPart ())
        return (GNL_OK);

    /*
        fprintf (stderr, " HASH ISO LIST:\n");
        for (i=0; i<BListSize (*HashListIsoGnls); i++)
        {
            IsoList = (BLIST)BListElt (*HashListIsoGnls, i);
            fprintf (stderr, " ISO = %d, SIZE = %d\n", i, BListSize (IsoList));
        }
    */

    return (GNL_OK);
}

/*-----*/
/* GnlDuplicateIsoNode */
/*-----*/
/* This procedure duplicates the nodes tree from the reference node */
/* 'NodeRef' and creates a new tree created in 'GnlIso' depending on the */
/* variables of it. */
/*-----*/
GNL_STATUS GnlDuplicateIsoNode (NodeRef, GnlIso, NewNode)
    GNL_NODE NodeRef;
    GNL_NODE GnlIso;
    GNL_NODE *NewNode;
{
    int i;
    GNL_VAR VarRef;
    GNL_VAR VarIso;
    GNL_NODE SonI;
    GNL_NODE NewSonI;
    BLIST NewList;

    if (GnlNodeOp (NodeRef) == GNL_VARIABLE)
    {
        VarRef = (GNL_VAR)GnlNodeSons (NodeRef);
        VarIso = (GNL_VAR)GnlVarHook (VarRef);

        if (GnlCreateNodeForVar (GnlIso, VarIso, NewNode))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    if (GnlNodeOp (NodeRef) == GNL_CONSTANTE)
    {
        if (GnlNodeSons (NodeRef))
        {
            if (GnlCreateNodeVdd (GnlIso, NewNode))
                return (GNL_MEMORY_FULL);
        }
        else
```

gnlopt.c

```
{  
    if (GnlCreateNodeVss (GnlIso, NewNode))  
        return (GNL_MEMORY_FULL);  
    }  
    return (GNL_OK);  
}  
  
if (GnlCreateNode (GnlIso, GnlNodeOp (NodeRef), NewNode))  
    return (GNL_MEMORY_FULL);  
  
if (BListCreateWithSize (BListSize (GnlNodeSons (NodeRef)),  
                        &NewList))  
    return (GNL_MEMORY_FULL);  
  
SetGnlNodeSons (*NewNode, NewList);  
  
for (i=0; i<BListSize (GnlNodeSons (NodeRef)); i++)  
{  
    SonI = (GNL_NODE)BListElt (GnlNodeSons (NodeRef), i);  
    if (GnlDuplicateIsoNode (SonI, GnlIso, &NewSonI))  
        return (GNL_MEMORY_FULL);  
    if (BListAddElt (NewList, (int)NewSonI))  
        return (GNL_MEMORY_FULL);  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlDuplicateListIsoGnls */  
/*-----*/  
/* This procedure will duplicates the nodes network presents in the */  
/* referenced Gnl 'GnlRef' and replaces the leaves (inputs var and */  
/* locals var) by the one in the 'GnlIso'. */  
/* Outputs in 'GnlRef' and 'GnlIso' are in the same order and have a 1 */  
/* to 1 mapping. */  
/*-----*/  
GNL_STATUS GnlDuplicateNodesInGnl (OriginalGnl, GnlRef, ListRefInputs,  
                                    ListRefLocals, ListRefOutputs, GnlIso)  
    GNL          OriginalGnl;  
    GNL          GnlRef;  
    BLIST        ListRefInputs;  
    BLIST        ListRefLocals;  
    BLIST        ListRefOutputs;  
    GNL          GnlIso;  
{  
    int           i;  
    BLIST        ListIsoInputs;  
    BLIST        ListIsoLocals;  
    BLIST        ListIsoOutputs;  
    BLIST        ListIsoFunctions;  
    GNL_VAR      VarRefI;  
    GNL_VAR      VarIsoI;  
    GNL_FUNCTION FunctionRefI;  
    GNL_FUNCTION FunctionIsoI;  
    GNL_NODE     NodeRefI;
```

```

GNL_NODE      NewNode;
BLIST         newList;
GNL_VAR       NewVar;
GNL_FUNCTION  NewFunction;

ListIsoInputs = GnlInputs (GnlIso);
ListIsoLocals = GnlLocals (GnlIso);
ListIsoOutputs = GnlOutputs (GnlIso);
ListIsoFunctions = GnlFunctions (GnlIso);

for (i=0; i<BListSize (GnlFunctions (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlFunctions (GnlRef), i);
    SetGnlVarHook (VarRefI, NULL);
}

/* We make the links between the variables of the reference Gnl and */
/* the isomorphic one.                                              */
/* we link the inputs                                              */
for (i=0; i<BListSize (ListRefInputs); i++)
{
    VarRefI = (GNL_VAR)BListElt (ListRefInputs, i);
    VarIsoI = (GNL_VAR)BListElt (ListIsoInputs, i);
    SetGnlVarHook (VarRefI, VarIsoI);
}
/* we link the locals.                                              */
for (i=0; i<BListSize (ListRefLocals); i++)
{
    VarRefI = (GNL_VAR)BListElt (ListRefLocals, i);
    VarIsoI = (GNL_VAR)BListElt (ListIsoLocals, i);
    SetGnlVarHook (VarRefI, VarIsoI);
}
/* we link the outputs.                                              */
for (i=0; i<BListSize (ListRefOutputs); i++)
{
    VarRefI = (GNL_VAR)BListElt (ListRefOutputs, i);
    VarIsoI = (GNL_VAR)BListElt (GnlOutputs (GnlIso), i);
    SetGnlVarHook (VarRefI, VarIsoI);
}

/* There may be extra variables which have been generated during the */
/* optimization.                                                       */
for (i=0; i<BListSize (GnlLocals (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlLocals (GnlRef), i);
    if (GnlVarHook (VarRefI) == NULL)
    {
        if (GnlCreateUniqueVarWithNoTestGnlId (OriginalGnl,
                                                GnlIso, "\\$D", &NewVar))
            return (GNL_MEMORY_FULL);
        if (GnlFunctionCreate (GnlIso, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);
        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionVar (NewFunction, NewVar);
    }
}

```

```

        SetGnlVarHook (VarRefI, NewVar);
    }
}

/* we duplicate equations for the corresponding outputs.      */
for (i=0; i<BListSize (GnlOutputs (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlOutputs (GnlRef), i);
    VarIsoI = (GNL_VAR)BListElt (GnlOutputs (GnlIso), i);

    FunctionRefI = GnlVarFunction (VarRefI);
    FunctionIsoI = GnlVarFunction (VarIsoI);

    NodeRefI = GnlFunctionOnSet (FunctionRefI);
    if (GnlDuplicateIsoNode (NodeRefI, GnlIso, &NewNode))
        return (GNL_MEMORY_FULL);

    SetGnlFunctionOnSet (FunctionIsoI, NewNode);
}

/* we duplicate equations for the corresponding locals.      */
for (i=0; i<BListSize (GnlLocals (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlLocals (GnlRef), i);
    VarIsoI = (GNL_VAR)GnlVarHook (VarRefI);

    FunctionRefI = GnlVarFunction (VarRefI);
    FunctionIsoI = GnlVarFunction (VarIsoI);

    NodeRefI = GnlFunctionOnSet (FunctionRefI);
    if (GnlDuplicateIsoNode (NodeRefI, GnlIso, &NewNode))
        return (GNL_MEMORY_FULL);

    SetGnlFunctionOnSet (FunctionIsoI, NewNode);
}

BSIZE (ListIsoInputs) = 0;
BSIZE (ListIsoLocals) = 0 ;
BSIZE (ListIsoFunctions) = 0;

/* Recreating the inputs list from the input list of the reference */
/* Gnl.                                                       */
for (i=0; i<BListSize (GnlInputs (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlInputs (GnlRef), i);
    VarIsoI = (GNL_VAR)GnlVarHook (VarRefI);
    if (BListAddElt (ListIsoInputs, (int)VarIsoI))
        return (GNL_MEMORY_FULL);
}

/* Recreating the locals list from the locals list of the reference */
/* Gnl.                                                       */
for (i=0; i<BListSize (GnlLocals (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlLocals (GnlRef), i);
    VarIsoI = (GNL_VAR)GnlVarHook (VarRefI);
    if (BListAddElt (ListIsoLocals, (int)VarIsoI))

```

```

        return (GNL_MEMORY_FULL);
    }

/* Recreating the Functions list from the Functions list of the      */
/* reference Gnl.                                                 */
for (i=0; i<BListSize (GnlFunctions (GnlRef)); i++)
{
    VarRefI = (GNL_VAR)BListElt (GnlFunctions (GnlRef), i);
    VarIsoI = (GNL_VAR)GnlVarHook (VarRefI);
    if (BListAddElt (ListIsoFunctions, (int)VarIsoI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
/* GnlDuplicateListIsoGnls                                     */
/*-----*/
/* we duplicate the optimized referenced Gnls and make a copy of it for */
/* each isomorphic Gnl.                                         */
/*-----*/
GNL_STATUS GnlDuplicateListIsoGnls (OriginalGnl, HashListIsoGnls)
    GNL          OriginalGnl;
    BLIST        HashListIsoGnls;
{
    int           i;
    int           j;
    BLIST        IsoList;
    GNL          RefGnl;
    GNL          GnlIsoJ;
    BLIST        ListRefInputs;
    BLIST        ListRefLocals;
    BLIST        ListRefOutputs;

/* The user asked for now folding of iso logic partitions so we have */
/* nothing to duplicate since each sub list has 1 single Gnl. */
if (!GnlEnvFoldIsoPart ())
    return (GNL_OK);

fprintf (stderr, " Duplicate List Gnls...\n");

for (i=0; i<BListSize (HashListIsoGnls); i++)
{
    j = 0;
    fprintf (stderr,
              "%c      o Duplicating List Iso [%d/%d] (Nb=%d)", 13, i,
              BListSize (HashListIsoGnls)-1, j+1);
    fflush (stderr);

    IsoList = (BLIST)BListElt (HashListIsoGnls, i);

    if (BListSize (IsoList) == 1)
        continue;
}

```

```

/* First Gnl is the referenced Gnl. */  

RefGnl = (GNL)BListElt (IsoList, 0);  

ListRefInputs = (BLIST)BlistElt (IsoList, 1);  

ListRefLocals = (BLIST)BlistElt (IsoList, 2);  

ListRefOutputs = (BLIST)BListElt (IsoList, 3);  
  

for (j=4; j<BListSize (IsoList); j++)  

{  

    GnlIsoJ = (GNL)BListElt (IsoList, j);  
  

    fprintf (stderr,  

            "%c          o Duplicating List Iso [%d/%d] (Nb=%d) ",  

            13, i, BListSize (HashListIsoGnls)-1, j-2);  

    fflush (stderr);  
  

    if (GnlDuplicateNodesInGnl (OriginalGnl, RefGnl,  

                               ListRefInputs, ListRefLocals, ListRefOutputs,  

                               GnlIsoJ))  

        return (GNL_MEMORY_FULL);  

}  

}  
  

fprintf (stderr, "\n");  

return (GNL_OK);
}  
  

/*-----*/  

/* GnlRebuildListGnls */  

/*-----*/  

/* This procedure rebuilds a new list 'ListGnls' from the hash list of */  

/* isomorphic Gnls and also deletes the hash list 'HashListIsoGnls' at */  

/* the end. */  

/*-----*/  

GNL_STATUS GnlRebuildListGnls (HashListIsoGnls, ListGnls)  

    BLIST      HashListIsoGnls;  

    BLIST      *ListGnls;  

{  

    int           i;  

    BLIST      IsoList;  

    int           j;  

    GNL       GnlJ;  

    BLIST      NewList;  
  

    BListQuickDelete (ListGnls);  

    if (BListCreate (ListGnls))  

        return (GNL_MEMORY_FULL);  
  

    for (i=0; i<BListSize (HashListIsoGnls); i++)  

    {  

        IsoList = (BLIST)BListElt (HashListIsoGnls, i);  

        if (BListSize (IsoList) == 1)  

        {  

            if (BListAppend (*ListGnls, &IsoList))  

                return (GNL_MEMORY_FULL);  

        }
    }
}
```

gnlopt.c

```
        return (GNL_MEMORY_FULL);
    continue;
}

GnlJ = (GNL)BListElt (IsoList, 0);
if (BListAddElt (*ListGnls, (int)GnlJ))
    return (GNL_MEMORY_FULL);
NewList = (BLIST)BListElt (IsoList, 1);
BListQuickDelete (&NewList);
NewList = (BLIST)BListElt (IsoList, 2);
BListQuickDelete (&NewList);
NewList = (BLIST)BListElt (IsoList, 3);
BListQuickDelete (&NewList);
for (j=4; j<BListSize (IsoList); j++)
{
    GnlJ = (GNL)BListElt (IsoList, j);
    if (BListAddElt (*ListGnls, (int)GnlJ))
        return (GNL_MEMORY_FULL);
}

BListQuickDelete (&IsoList);
}

BListQuickDelete (&HashListIsoGnls);

return (GNL_OK);
}

/*-----*/
/* GnlSimplifyWithVssVddOnNode */
/*-----*/
/* This procedure returns a new GNL_NODE thru the variable 'NewNode' */
/* which is the simplification the node tree 'Node' by using Boolean */
/* relations like 0.X --> 0, 1.X --> X, ... */
/*-----*/
GNL_STATUS GnlSimplifyWithVssVddOnNode (Gnl, Node, NewNode)
    GNL             Gnl;
    GNL_NODE Node;
    GNL_NODE *NewNode;
{
    int             i;
    GNL_NODE SonI;
    GNL_NODE NewSonI;

switch (GnlNodeOp (Node)) {
    case GNL_VARIABLE:
        *NewNode = Node;
        return (GNL_OK);

    case GNL_AND:
        for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
        {
            SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
            if (GnlSimplifyWithVssVddOnNode (Gnl, SonI,
                &NewSonI))

```

```

        return (GNL_MEMORY_FULL);
    BListElt (GnlNodeSons (Node), i) = (int)NewSonI;
    SonI = NewSonI;
    if (GnlNodeIsVss (SonI))
    {
        *NewNode = SonI;
        return (GNL_OK);
    }
    if (GnlNodeIsVdd (SonI))
    {
        if (BListSize (GnlNodeSons (Node)) == 1)
        {
            *NewNode = SonI;
            return (GNL_OK);
        }
        BListDelInsert (GnlNodeSons (Node), i+1);
        i--;
        continue;
    }
}
if (BListSize (GnlNodeSons (Node)) == 1)
{
    *NewNode = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);
    return (GNL_OK);
}
*NewNode = Node;
break;

case GNL_OR:
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlSimplifyWithVssVddOnNode (Gnl, SonI,
                                         &NewSonI))
            return (GNL_MEMORY_FULL);
        BListElt (GnlNodeSons (Node), i) = (int)NewSonI;
        SonI = NewSonI;
        if (GnlNodeIsVdd (SonI))
        {
            *NewNode = SonI;
            return (GNL_OK);
        }
        if (GnlNodeIsVss (SonI))
        {
            if (BListSize (GnlNodeSons (Node)) == 1)
            {
                *NewNode = SonI;
                return (GNL_OK);
            }
            BListDelInsert (GnlNodeSons (Node), i+1);
            i--;
            continue;
        }
    }
}
if (BListSize (GnlNodeSons (Node)) == 1)
{
    *NewNode = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);
}

```

```

        return (GNL_OK);
    }
    *NewNode = Node;
    break;

case GNL_NOT:
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);
    if (GnlSimplifyWithVssVddOnNode (Gnl, SonI,
                                      &NewSonI))
        return (GNL_MEMORY_FULL);
    BListElt (GnlNodeSons (Node), 0) = (int)NewSonI;
    SonI = NewSonI;
    if (GnlNodeIsVss (SonI))
    {
        if (GnlCreateNodeVdd (Gnl, NewNode))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }
    if (GnlNodeIsVdd (SonI))
    {
        if (GnlCreateNodeVss (Gnl, NewNode))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    *NewNode = Node;
    break;

case GNL_CONSTANTE:
    *NewNode = Node;
    return (GNL_OK);

default:
    fprintf (stderr, " ERROR: Node operator is unknown\n");
    exit (1);
}

return (GNL_OK);
}

/*
 * GnlSimplifyWithVssVdd
 */
/*
 * This procedure performs simplifications when encountering special
 * signals which are VDD or VSS in the GNL_NODE trees.
 */
GNL_STATUS GnlSimplifyWithVssVdd (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_VAR      VarI;
    GNL_FUNCTION FunctionI;
    GNL_NODE     NodeI;
    GNL_NODE     NewNode;

```

gnlopt.c

```
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (Vari);
    NodeI = GnlFunctionOnSet (FunctionI);
    if (GnlSimplifyWithVssVddOnNode (Gnl, NodeI, &NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlFunctionOnSet (FunctionI, NewNode);
}

return (GNL_OK);
}

/*-----
/* GnlLimitDepthFunctionsOnNode
 */
-----*/
GNL_STATUS GnlLimitDepthFunctionsOnNode (Gnl, Node, Depth, NewNode)
    GNL          Gnl;
    GNL_NODE     Node;
    int          Depth;
    GNL_NODE *NewNode;
{
    GNL_VAR      NewVar;
    GNL_FUNCTION NewFunction;
    GNL_NODE     NodeI;
    GNL_NODE     NewNodeI;
    int          i;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        *NewNode = Node;
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        *NewNode = Node;
        return (GNL_OK);
    }

    if (Depth % 6 == 0)
    {
        if (GnlCreateUniqueVar (Gnl, "X", &NewVar))
            return (GNL_MEMORY_FULL);

        if (BListAddElt (GnlLocals(Gnl), (int)NewVar))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (Gnl, GnlNbLocal(Gnl)+1);

        if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
            return (GNL_MEMORY_FULL);

        SetGnlVarFunction (NewVar, NewFunction);
        SetGnlFunctionOnSet (NewFunction, Node);

        if (BListAddElt (GnlFunctions(Gnl), (int)NewVar))
```

gnlopt.c

```
        return (GNL_MEMORY_FULL);

    if (GnlCreateNodeForVar (Gnl, NewVar, NewNode))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlLimitDepthFunctionsOnNode (Gnl, NodeI, Depth+1, &NewNodeI))
        return (GNL_MEMORY_FULL);
    BListElt (GnlNodeSons (Node), i) = (int)NewNodeI;
}

*NewNode = Node;

return (GNL_OK);
}

/*-----*/
/* GnlLimitDepthFunctions */
/*-----*/
GNL_STATUS GnlLimitDepthFunctions (Gnl)
    GNL          Gnl;
{
    int           i;
    GNL_VAR      VarI;
    GNL_NODE     Node;
    GNL_NODE     NewNode;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        if (!GnlVarFunction (VarI))
            continue;

        Node = GnlFunctionOnSet (GnlVarFunction (VarI));
        if (GnlLimitDepthFunctionsOnNode (Gnl, Node, 1, &NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlFunctionOnSet (GnlVarFunction (VarI), NewNode);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlOptimize */
/*-----*/
GNL_STATUS GnlOptimize (Gnl, OptPanel, NewGnl)
```

gnlopt.c

```
GNL          Gnl;
LS_OPT_PANEL OptPanel;
GNL          *NewGnl;
{
    int          i;
    GNL          GnlI;
    BLIST      ListGnls;
    GNL_VAR    VarI;
    int          Flattened;
    GNL_INFO   BeforeGnlInfo;
    GNL_INFO   AfterGnlInfo;
    GNL_INFO   GnlInfo;
    float       DiffLitt;
    float       DiffDepth;
    float       ImproveLitt;
    float       ImproveDepth;
    BLIST      IsoGnls;
    BLIST      HashListIsoGnls;

    G_OptPanel = OptPanel;
    *NewGnl= Gnl;

    fprintf (stderr, " Constant Pushing in [%s]...\n", GnlName (Gnl));
    if (GnlSimplifyWithVssVdd (Gnl))
        return (GNL_MEMORY_FULL);

    fprintf (stderr, " Collapsing glue logic in [%s]...\n", GnlName (Gnl));
    if (GnlInjectSingleVar (Gnl))
        return (GNL_MEMORY_FULL);

    if (GnlEnvDontTouch ())
    {
        *NewGnl= Gnl;
        return (GNL_OK);
    }

/* According to the value of 'LsOptMaxInject' either we do an */
/* automatic injection (value is -1) or we do a user injection and */
/* we use the value of 'LsOptMaxInject'. */
if (LsOptPanelDoOpt (OptPanel) && (LsOptMaxInject (OptPanel) == -1))
{
    fprintf (stderr,
    " Selective re-injection of logic in [%s](Inject=Default)...\\n",
    GnlName (Gnl));
    if (GnlPerformAutomaticInject (Gnl, OptPanel))
        return (GNL_MEMORY_FULL);
}
else
{
    fprintf (stderr,
    " Selective re-injection of logic in [%s](Inject=%d)...\\n",
    GnlName (Gnl), LsOptMaxInject (OptPanel));
    if (GnlInject (Gnl, LsOptMaxInject (OptPanel)))
        return (GNL_MEMORY_FULL);
```

```

}

if (LsOptPanelDoOpt (OptPanel))
{
    fprintf (stderr, " Propagating inverters...\n");
    if (GnlPropagateInv (Gnl))
        return (GNL_MEMORY_FULL);
}

if (GnlGetGnlInfo (Gnl, &BeforeGnlInfo))
    return (GNL_MEMORY_FULL);

/* No optimization to do. */
if (!LsOptPanelDoOpt (OptPanel))
{
    fprintf (stderr, "      o LITERALS      = %d\n",
            GnlInfoNbLit (BeforeGnlInfo));
    fprintf (stderr, "      o MAX. COMB. PATH = %d\n",
            GnlInfoMaxDepth (BeforeGnlInfo));
    *NewGnl = Gnl;
    return (GNL_OK);
}

fprintf (stderr, "      o LITERALS      = %d\n",
        GnlInfoNbLit (BeforeGnlInfo));
fprintf (stderr, "      o MAX. COMB. PATH = %d\n",
        GnlInfoMaxDepth (BeforeGnlInfo));

fprintf (stderr, " Partitioning Glue Logic... \n", i);
if (GnlPartitionGnlQuick (Gnl, &ListGnls))
    return (GNL_MEMORY_FULL);

if (!BListSize (ListGnls)) /* Actually no partitions */ 
{
    *NewGnl = Gnl;
    return (GNL_OK);
}

/* we sort the Gnl in the list 'ListGnls' according to their size */
/* in term of litterals. */
fprintf (stderr, " Sorting Partitions...\n");
GnlSortListOfGnl (ListGnls);

if (GnlHashListIsomorphGnl (ListGnls, &HashListIsoGnls))
    return (GNL_MEMORY_FULL);

/* we sort the iso gnls sub-lists in the decreasing order of the */
/* NbLit (RefGnl)*BListSize (sub-list). */
GnlSortListOfHashListIsoGnls (HashListIsoGnls);

/* For each partitions ... */
for (i=0; i<BListSize (HashListIsoGnls); i++)
{
    IsoGnls = (BLIST)BListElt (HashListIsoGnls, i);
    GnLI = (GNL)BListElt (IsoGnls, 0);

    fprintf (stderr,

```

gnlopt.c

```
%c Optimization [%d/%d] [%d] (IN=%d,OUT=%d,LOC=%d,LIT=%d) -Flatten      ",
    13, i, BListSize (HashListIsoGnls),
    BListSize (IsoGnls), GnlNbIn (GnLI),
    GnlNbOut (GnLI),
    GnlNbLocal (GnLI), GnlNbLitt (GnLI));
fflush (stderr);

if (GnlNbIn (GnLI)+GnlNbOut (GnLI)+GnlNbLocal (GnLI) > 1200)
    continue;

/* Flattening with control value.                                     */
if (GnlMakeGnlFlattenable (GnLI,
                           LsOptMaxFlattenCubes (OptPanel),
                           &Flattened))
    return (GNL_MEMORY_FULL);

/* eventually redo the flattening ...                                */
if (!Flattened)
{
    fprintf (stderr,
             "\n WARNING: Restructuring previous logic re-injection\n");
    fprintf (stderr,
             "           Re-injection value was <%d>\n",
             LsOptMaxInject (OptPanel));
    if (GnlComputeCubesInGnl (GnLI))
        return (GNL_MEMORY_FULL);
}

fprintf (stderr,
%c Optimization [%d/%d] [%d] (IN=%d,OUT=%d,LOC=%d,LIT=%d) -Minimize      ",
    13, i, BListSize (HashListIsoGnls),
    BListSize (IsoGnls), GnlNbIn (GnLI),
    GnlNbOut (GnLI),
    GnlNbLocal (GnLI), GnlNbLitt (GnLI));
fflush (stderr);

/* 2-level minimization                                         */
if (LsMinGnl (GnLI, 0))
    return (GNL_MEMORY_FULL);

fprintf (stderr,
%c Optimization [%d/%d] [%d] (IN=%d,OUT=%d,LOC=%d,LIT=%d) -Factorize      ",
    13, i, BListSize (HashListIsoGnls),
    BListSize (IsoGnls), GnlNbIn (GnLI),
    GnlNbOut (GnLI),
    GnlNbLocal (GnLI), GnlNbLitt (GnLI));
fflush (stderr);

/* Algebraic factorization + common part extraction               */
if (LsFactorize (Gnl, GnLI, OptPanel))
    return (GNL_MEMORY_FULL);

}

fprintf (stderr,
%c Logic Optimization [%d/%d]                                         \n",
    13, i, BListSize (HashListIsoGnls),
    BListSize (IsoGnls), GnlNbIn (GnLI),
    GnlNbOut (GnLI),
    GnlNbLocal (GnLI), GnlNbLitt (GnLI));
fflush (stderr);
```

gnlopt.c

```
    13, i, BListSize (HashListIsoGnls));
fflush (stderr);

/* we duplicate the optimized referenced Gnls and make a copy of it */
/* for each isomorphic Gnl.                                         */
if (GnlDuplicateListIsoGnls (Gnl, HashListIsoGnls))
    return (GNL_MEMORY_FULL);

/* We rebuild the list of Gnls from the hash list of iso Gnls. */
fprintf (stderr, " Rebuild List Gnls...\n");
if (GnlRebuildListGnls (HashListIsoGnls, &ListGnls))
    return (GNL_MEMORY_FULL);

/* Merging of all the partitions.                                     */
fprintf (stderr, " Merging Partitions...\n");
if (GnlMergeGnl (ListGnls, NewGnl))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " Collapsing glue logic...\n");
if (GnlInjectSingleVar (*NewGnl))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " Extracting New Infos...\n");
if (GnlGetGnlInfo (*NewGnl, &AfterGnlInfo))
    return (GNL_MEMORY_FULL);

DiffLitt = (float)GnlInfoNbLit (BeforeGnlInfo) -
           (float)GnlInfoNbLit (AfterGnlInfo);
if (GnlInfoNbLit (BeforeGnlInfo))
    ImproveLitt = (100*(float)DiffLitt)/
                   (float)GnlInfoNbLit (BeforeGnlInfo);
else
    ImproveLitt = 0.0;
DiffDepth = (float)GnlInfoMaxDepth (BeforeGnlInfo) -
            (float)GnlInfoMaxDepth (AfterGnlInfo);
if (GnlInfoMaxDepth (BeforeGnlInfo))
    ImproveDepth = (100*(float)DiffDepth)/
                   (float)GnlInfoMaxDepth (BeforeGnlInfo);
else
    ImproveDepth = 0.0;

fprintf (stderr,
        "          O LITERALS:      %6d --> %6d [Improve: %4.1f %c]\n",
        GnlInfoNbLit (BeforeGnlInfo), GnlInfoNbLit (AfterGnlInfo),
        ImproveLitt, '%');
fprintf (stderr,
        "          O MAX.COMB.PATH: %6d --> %6d [Improve: %4.1f %c]\n",
        GnlInfoMaxDepth (BeforeGnlInfo),
        GnlInfoMaxDepth (AfterGnlInfo), ImproveDepth, '%');

fprintf (stderr, "\n");

return (GNL_OK);
}

/*-----*/
/* GnlSynthesizeNetworkRec                                         */
/*-----*/
```

gnlopt.c

```
/*-----*/
GNL_STATUS GnlSynthesizeNetworkRec (Nw, Gnl, GnlLibc, OutFile, Panel)
    GNL_NETWORK Nw;
    GNL          *Gnl;
    LIBC_LIB GnlLibc;
    FILE         *OutFile;
    LS_OPT_PANEL Panel;
{
    int             i;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL           GnlCompoI;
    GNL           OptGnl;
    BLIST          Components;
    int            Done;
    GNL_INFO       GnlInfo;

    /* We already synthesized this current '*Gnl'. The results has been */
    /* stored previously in 'GnlSynthesizedGnl (Gnl)'.                      */
    if (GnlTag (*Gnl) == GnlNetworkTag (Nw))
    {
        *Gnl = GnlSynthesizedGnl (*Gnl);
        return (GNL_OK);
    }

    SetGnlTag (*Gnl, GnlNetworkTag (Nw));

    Components = GnlComponents (*Gnl);

    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;
        UserCompoI = (GNL_USER_COMPONENT)ComponentI;
        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

        if (!GnlCompoI)
            continue;

        if (GnlSynthesizeNetworkRec (Nw, &GnlCompoI, GnlLibc, OutFile,
                                      Panel))
            return (GNL_MEMORY_FULL);
        SetGnlUserComponentGnlDef (UserCompoI, GnlCompoI);
    }

    fprintf (stderr, "\n ======");
    for (i=0; i<strlen (GnlName (*Gnl)); i++)
        fprintf (stderr, "=");
    fprintf (stderr, "\n");
    fprintf (stderr, " SYNTHESIZING [%s] \n",
            GnlName (*Gnl));
    fprintf (stderr, " ======");
    for (i=0; i<strlen (GnlName (*Gnl)); i++)
        fprintf (stderr, "=");
}
```

```

if (GnlGetGnlInfo (*Gnl, &GnlInfo))
    return (GNL_MEMORY_FULL);

fprintf (stderr, "\n      o NB. INPUTS          = %d\n",
         GnlInfoNbInputs (GnlInfo));
fprintf (stderr, "      o NB. OUTPUTS        = %d\n",
         GnlInfoNbOutputs (GnlInfo));
fprintf (stderr, "      o NB. INOUTS        = %d\n",
         GnlInfoNbInOuts (GnlInfo));
fprintf (stderr, "      o NB. LITERALS       = %d\n",
         GnlInfoNbLit (GnlInfo));
fprintf (stderr, "      o NB. FLIP-FLOPS     = %d\n",
         GnlInfoNbFlipFlops (GnlInfo));
fprintf (stderr, "      o NB. LATCHES        = %d\n",
         GnlInfoNbLatches (GnlInfo));
fprintf (stderr, "      o NB. TRISTATES      = %d\n",
         GnlInfoNbTriStates (GnlInfo));
fprintf (stderr, "      o NB. BUFFERS        = %d\n",
         GnlInfoNbBuffers (GnlInfo));
fprintf (stderr, "      o MAX. COMBINATORIAL PATH = %d\n",
         GnlInfoMaxDepth (GnlInfo));

/* We map the sequential elements. */ */
if (GnlMapSequential (*Gnl, GnlLibc))
    return (GNL_MEMORY_FULL);

/* We map the 3-states elements. */ */
if (GnlMap3State (*Gnl, GnlLibc))
    return (GNL_MEMORY_FULL);

/* Invoke the logic optimizer on the module 'Gnl'. */ */
if (GnlOptimize (*Gnl, Panel, &OptGnl))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " Constant Pushing in [%s]...\n", GnlName (OptGnl));
if (GnlSimplifyWithVssVdd (OptGnl))
    return (GNL_MEMORY_FULL);

/* Invoke the Technology mapper on the module 'Gnl'. */ */
/* The result can be eventually dumped in the file 'OutFile'. */
if (GnlMap (Nw, OptGnl, GnlLibc, OutFile, LsOptPanelMapEffort (Panel),
            LsOptPanelCriter (Panel), &Done))
    return (GNL_MEMORY_FULL);

/* We store the result of the synthesis in the given '**Gnl'. */ */
SetGnlSynthesizedGnl (*Gnl, OptGnl);

/* we return the new synthesized Gnl. */ */
*Gnl = OptGnl;

SetGnlNbDffs (*Gnl, GnlInfoNbFlipFlops (GnlInfo));
SetGnlNbLatches (*Gnl, GnlInfoNbLatches (GnlInfo));
SetGnlNbTristates (*Gnl, GnlInfoNbTriStates (GnlInfo));

return (GNL_OK);

```

gnlopt.c

```
}

/*-----
/* GnlGetNetworkPredefinedComponentsRec
/*-----*/
void GnlGetNetworkPredefinedComponentsRec (Gnl, NbDffs, NbLatches,
                                             NbTristates, NbBuffers)
{
    GNL          Gnl;
    int          *NbDffs;
    int          *NbLatches;
    int          *NbTristates;
    int          *NbBuffers;
{
    int          i;
    BLIST        Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL          GnlCompoI;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;

    Components = GnlComponents (Gnl);

    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);
        switch (GnlComponentType (ComponentI)) {
            case GNL_SEQUENTIAL_COMPO:
                SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
                if (GnlSeqComponentIsDFF (SeqCompo))
                    (*NbDffs)++;
                else
                    (*NbLatches)++;
                break;

            case GNL_TRISTATE_COMPO:
                (*NbTristates)++;
                break;

            case GNL_USER_COMPO:
                UserCompoI = (GNL_USER_COMPONENT)ComponentI;
                GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

                if (GnlCompoI)
                    GnlGetNetworkPredefinedComponentsRec (GnlCompoI,
                                                        NbDffs, NbLatches, NbTristates,
                                                        NbBuffers);
                break;

            case GNL_BUF_COMPO:
                (*NbBuffers)++;
                break;

        }
    }
}
```

gnlopt.c

```
/*-----*/
/* GnlGetNetworkPredefinedComponents */
/*-----*/
/* This procedure extracts the number of predefined components (Dffs,    */
/* Latches, Tristates, Buffers) which are in the network.          */
/*-----*/
void GnlGetNetworkPredefinedComponents (Nw, NbDffs, NbLatches,
                                         NbTristates, NbBuffers)
{
    GNL_NETWORK Nw;
    int          *NbDffs;
    int          *NbLatches;
    int          *NbTristates;
    int          *NbBuffers;
}

float      Area;
GNL        TopGnl;

TopGnl = GnlNetworkTopGnl (Nw);

*NbDffs = *NbLatches = *NbTristates = *NbBuffers = 0;
GnlGetNetworkPredefinedComponentsRec (TopGnl, NbDffs, NbLatches,
                                       NbTristates, NbBuffers);
}

/*-----*/
/* GnlSynthesizeNetwork */
/*-----*/
/* Main procedure doing Optimization and mapping of the whole network. */
/* The final verilog description is dumped into the file 'OutFile'. */
/*-----*/
GNL_STATUS GnlSynthesizeNetwork (Nw, GnlLibc, OutFile, Panel)
{
    GNL_NETWORK Nw;
    LIBC_LIB GnlLibc;
    FILE        *OutFile;
    LS_OPT_PANEL Panel;
}

GNL        TopGnl;
double    Area;
int       Depth;
GNL_LIB   HGnlLib;
int       NbDffs;
int       NbLatches;
int       NbTristates;
int       NbBuffers;

HGnlLib = (GNL_LIB)LibHook (GnlLibc);

/* Printing the header of the output file */
GnlPrintVerilogSimpleNwHeader (OutFile, Nw, HGnlLib);
                                         */

GnlResetGnlNetworkTag (Nw);
```

gnlopt.c

```
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

TopGnl = GnlNetworkTopGnl (Nw);
if (GnlSynthesizeNetworkRec (Nw, &TopGnl, GnlLibc, OutFile, Panel))
    return (GNL_MEMORY_FULL);
SetGnlNetworkTopGnl (Nw, TopGnl);

Area = GnlGetNetworkGateArea (Nw);
SetGnlNetworkArea (Nw, Area);

GnlGetNetworkPredefinedComponents (Nw, &NbDffs, &NbLatches,
                                  &NbTristates, &NbBuffers);
SetGnlNetworkNbDffs (Nw, NbDffs);
SetGnlNetworkNbLatches (Nw, NbLatches);
SetGnlNetworkNbTristates (Nw, NbTristates);
SetGnlNetworkNbBuffers (Nw, NbBuffers);

return (GNL_OK);

}

/*----- EOF -----*/
```

DO NOT ALTER THIS FILE

gnlopt.h

```
/*-----*/
/*          File:          lsopt.h
/*          Version:        1.1
/*          Modifications: -
/*          Documentation: -
/*-----*/
#ifndef LSOPT_H
#define LSOPT_H

/*-----*/
/* LS_OPT_PANEL_REC
/*-----*/
typedef struct LS_OPT_PANEL_STRUCT {

    /* Parameters for Factorisation + Extraction */
    int      Criter;
    int      MapEffort;
    int      DoOptimisation;
    int      DoDivision;
    int      DoExtraction;
    int      NbMaxDivisionKernels;
    int      NbMaxExtractionKernels;
    GNL_STATUS (*DivGainFunc) ();
    GNL_STATUS (*DivChoiceFunc) ();
    GNL_STATUS (*ExtGainFunc) ();
    GNL_STATUS (*ExtChoiceFunc) ();

    /* Parameters for collapsing */
    int      MaxLitToInject;

    /* Parameters for logic reinjection */
    int      MaxInject;

    /* Parameters for 2-level minimization */
    int      MaxFlattenCubes;
}

LS_OPT_PANEL_REC, *LS_OPT_PANEL;

#define LsOptPanelCriter(p)      (((p)->Criter))
#define LsOptPanelMapEffort(p)   (((p)->MapEffort))
#define LsOptPanelDoOpt(p)       (((p)->DoOptimisation))
#define LsOptPanelDoDiv(p)       (((p)->DoDivision))
#define LsOptPanelDoExt(p)       (((p)->DoExtraction))
#define LsOptPanelMaxDivKer(p)   (((p)->NbMaxDivisionKernels))
#define LsOptPanelMaxExtKer(p)   (((p)->NbMaxExtractionKernels))
#define LsOptPanelDivGainFunc(p) (((p)->DivGainFunc))
#define LsOptPanelDivChoiceFunc(p) (((p)->DivChoiceFunc))
#define LsOptPanelExtGainFunc(p) (((p)->ExtGainFunc))
#define LsOptPanelExtChoiceFunc(p) (((p)->ExtChoiceFunc))
#define LsOptMaxLitToInject(p)   (((p)->MaxLitToInject))
#define LsOptMaxInject(p)        (((p)->MaxInject))
#define LsOptMaxFlattenCubes(p)  (((p)->MaxFlattenCubes))

#define SetLsOptPanelCriter(p,d) ((p)->Criter = d)
```

gnlopt.h

```
#define SetLsOptPanelMapEffort(p,d) (((p)->MapEffort = d))
#define SetLsOptPanelDoOpt(p,d) (( (p)->DoOptimisation = d))
#define SetLsOptPanelDoDiv(p,d) (( (p)->DoDivision = d))
#define SetLsOptPanelDoExt(p,d) (((p)->DoExtraction = d))
#define SetLsOptPanelMaxDivKer(p,n) (((p)->NbMaxDivisionKernels = n))
#define SetLsOptPanelMaxExtKer(p,n) (((p)->NbMaxExtractionKernels = n))
#define SetLsOptPanelDivGainFunc(p,f) (((p)->DivGainFunc = f))
#define SetLsOptPanelDivChoiceFunc(p,f) (((p)->DivChoiceFunc = f))
#define SetLsOptPanelExtGainFunc(p,f) (((p)->ExtGainFunc = f))
#define SetLsOptPanelExtChoiceFunc(p,f) (((p)->ExtChoiceFunc = f))
#define SetLsOptMaxLitToInject(p,m) (((p)->MaxLitToInject = m))
#define SetLsOptMaxInject(p,m) (((p)->MaxInject = m))
#define SetLsOptMaxFlattenCubes(p,m) (((p)->MaxFlattenCubes = m))

/*----- EOF -----*/
#endif
```

DRAFT RELEASE UNDER EMBARGO

gnloption.c

```
/*-----*/  
/*-----*/  
/*      File:          gnloption.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*-----*/  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "gnlopt.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnloption.h"  
  
#include "blist.e"  
  
/* ----- */  
/* functions called during the parsing of the command line. */  
/* ----- */  
void GnlModeOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlInputOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlOutputOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlLogOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlOptForceOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlCriterionOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlInjectOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlFlattenOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlFlattenStrOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlLibraryOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlTopOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlModelDirOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlModelExtOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlBuildLibForceOption (GNL_ENV conf, int argc, char **argv, int  
*arg_no);  
void GnlReferenceOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlMaxBddNodeOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlFlattenInstanceNameOption (GNL_ENV conf, int argc, char **argv, int  
*arg_no);  
void GnlInputFormatOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlFsmDirOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlPrintHierarchyOption (GNL_ENV conf, int argc, char **argv, int  
*arg_no);  
void GnlReportDataOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlPrintNbCritPath (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlFoldIsoPartOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlPrintMaxFanoutVarsOption (GNL_ENV conf, int argc, char **argv, int  
*arg_no);  
void GnlUseQBarOption (GNL_ENV conf, int argc, char **argv, int *arg_no);  
void GnlRespectLibConstraintsOption (GNL_ENV conf, int argc, char **argv, int  
*arg_no);
```

gnloption.c

```
void GnlPrintClkDomainOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlPrintCritRegionOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlMaxCellSizeSupportOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlMinShareLogicSizeOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlUseVerilogPrimitivesOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlIgnorePostOptOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlInputAssocOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlOutputAssocOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlVDDOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlXorUnprovedOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlIgnoreRandomSimOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlPinSeqPolMatchingOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlPrintResizedGatesOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlReplaceOnlyCombiCompoOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlRemoveBufferOption (GNL_ENV conf, int argc, char **argv, int *arg_no);
void GnlDontTouchOption (GNL_ENV conf, int argc, char **argv, int *arg_no);

/* -----
/* Table of options for HUBBLE
/* -----
#define GNL_NB_OPTIONS      42

GNL_OPTION_REC G_Options[GNL_NB_OPTIONS] =
{
{"-mode",
"-m",
"[synthesis|estimation|translate|verification|post_optimization]",
"Specify if HUBLEE-RTL is in Synthesis, in Estimation, Translation,",
", Verification or Post-Optimization Mode.",
NULL,
NULL,
NULL,
GnlModeOption,
GNL_OPTION_FILE
},
{"-input",
"-i",
"<String>",
"Specifies file to use as input.",
NULL,
NULL,
NULL,
NULL,
GnlInputOption,
GNL_OPTION_FILE
}
```

gnloption.c

```
},
{"-output",
"-o",
"<String>",
"Specifies file to use as output.",
NULL,
NULL,
NULL,
NULL,
GnlOutputOption,
GNL_OPTION_FILE
},
{"-logfile",
"-log",
"<String>",
"Specifies file to use as log file; default is \"stderr\".",
NULL,
NULL,
NULL,
NULL,
GnlLogOption,
GNL_OPTION_FILE
},
{"-input_format",
"-if",
"[fsm|vlg]",
"Specifies if the input format is .fsm (fsm) or verilog (vlg);",
"default is '\fsm\'",
NULL,
NULL,
NULL,
GnlInputFormatOption,
GNL_OPTION_FILE
},
{"-fsm_dir",
"-fd",
"<String>",
"Specifies the path where all the FSM files of the current",
"design are stored. Default is '.'.",
NULL,
NULL,
NULL,
GnlFsmDirOption,
GNL_OPTION_FILE
},
{"-optimisation_strength",
"-os",
"[low|medium|high]",
"Specifies strength with which logic optimization will be performed.",
NULL,
NULL,
NULL,
NULL,
GnlOptForceOption,
GNL_OPTION_FILE
},
{"-criterion",
```

gnloption.c

```
--c",
"[area_gate|area_net|nl_delay|timing|power]",
Specifies which criterion synthesis will optimize during Tech. mapping.,
Default is [area_gate]. This default option optimize the netlist in",
"term of gates area. [area_net] minimizes the number of nets thus",
"nets area. [timing] minimizes the longest path of nets between gates.",
"Finally, [power] optimize the dynamic power consumption.",
GnlCriterionOption,
GNL_OPTION_FILE
},
{"-replace_only_combinatorial_cells",
"-rocc",
"",
"Tell HUBBLE to replace only the combinational cells in the original",
"netlist by their functionality in order to do re-synthesis on the",
"combinational parts of the design.",
NULL,
NULL,
GnlReplaceOnlyCombiCompoOption,
GNL_OPTION_FILE
},
{"-dont_touch_netlist",
"-dtn",
"",
"Tell HUBBLE to not touch the original netlist and keeps its structure",
"as it is.",
NULL,
NULL,
NULL,
NULL,
GnlDontTouchOption,
GNL_OPTION_FILE
},
{"-remove_buffers",
"-rb",
"",
"Tell HUBBLE to remove the buffers in the original netlist.",
NULL,
NULL,
NULL,
NULL,
GnlRemoveBufferOption,
GNL_OPTION_FILE
},
{"-vdd",
"-vdd",
"<Integer>",
"Specifies the voltage of the design in mV.",
NULL,
NULL,
NULL,
NULL,
GnlVDDOption,
GNL_OPTION_FILE
},
{"-use_sequential_qbar",
"-usqb",
"",
"",

GnlSequentialQbarOption,
GNL_OPTION_FILE
},
```

gnloption.c

```
"Specifies if output Q bar is used for sequential elements.",  
NULL,  
NULL,  
NULL,  
NULL,  
NULL,  
GnlUseQBarOption,  
GNL_OPTION_FILE  
},  
{"-use_sequential_clock_polarity_matching",  
"-uscpm",  
"  
",  
"Tells HUBBLE to select in priority sequential elements with exact ",  
"clock polarity matching. This can avoid gated clock with inverters",  
"in front of the clock pin of the sequential element.",  
NULL,  
NULL,  
GnlPinSeqPolMatchingOption,  
GNL_OPTION_FILE  
},  
{"-max_cell_fanin",  
"-mcf",  
"<Integer>",  
"Specifies the maximum FanIn of library cells taken into account.",  
"A greater max FanIn can slower the technology mapping phase but",  
"may give better results. By default the value is 6 and can be",  
"increased to a maximum reasonable value of 8.",  
NULL,  
GnlMaxCellSizeSupportOption,  
GNL_OPTION_FILE  
},  
{"-minimum_shared_logic_size",  
"-msls",  
"<Integer>",  
"Specifies the minimum size of logic which will be shared during",  
"the optimization phase. Small size logic can influence badly the",  
"efficiency of the design because it involves generally a big FanOut.",  
"The size number is relative to the number of cubes composing the",  
"shared logic.",  
GnlMinShareLogicSizeOption,  
GNL_OPTION_FILE  
},  
{"-use_verilog_primitives",  
"-uvp",  
"  
",  
"Tells HUBBLE to print out an output netlist using Verilog primitives",  
"like 'or', 'and', 'buf', ... instead of using technology library cells.",  
"By default cells of the specified technology library are used.",  
NULL,  
NULL,  
GnlUseVerilogPrimitivesOption,  
GNL_OPTION_FILE  
},  
{"-ignore_library_constraints",  
"-ilc",  
"  
",  
"Tells HUBBLE to not take into account the library constraints (Max.",  
"capacitance, Max FanOut, Max Transition).",
```

gnloption.c

```
NULL,
NULL,
NULL,
GnlRespectLibConstraintsOption,
GNL_OPTION_FILE
},
{"-ignore_post_optimization",
"-ipo",
"",
"",
"Tells HUBBLE to not take into account the post optimization phase",
"done on the mapped netlist. Post optimization performs netlist",
"optimizations like buffer insertions, gates resizing, logic",
"restructuring.",
NULL,
GnlIgnorePostOptOption,
GNL_OPTION_FILE
},
{"-print_resized_gates",
"-prg",
"",
"",
"Tells HUBBLE to print out the gates resized during the",
"post-optimization phase.",
NULL,
NULL,
NULL,
GnlPrintResizedGatesOption,
GNL_OPTION_FILE
},
{"-print_clock_domain",
"-pcd",
"",
"",
"Tells HUBBLE to print out the clock domains of each clock.",
"This option is available only when -rd timings is invoked.",
NULL,
NULL,
NULL,
GnlPrintClkDomainOption,
GNL_OPTION_FILE
},
{"-print_epsilon_critical_region",
"-pecr",
"",
"",
"Tells HUBBLE to print out the Epsilon critical region representing",
"the number of critical gates compare to the total number of gates.",
"This option is available only when -rd timings is invoked.",
NULL,
NULL,
GnlPrintCritRegionOption,
GNL_OPTION_FILE
},
{"-lib",
"-l",
"<String>",
"Specifies the target technology library (.lib format).",
NULL,
NULL,
NULL,
```

```

NULL,
GnlLibraryOption,
GNL_OPTION_FILE
},
{"-inject",
"-inj",
"<Integer>",
"Specifies degree of logic reinjection performed. Greater is the",
"integer and stronger is the flattening. Default is -1.",
NULL,
NULL,
NULL,
GnlInjectOption,
GNL_OPTION_FILE
},
{"-flat_hierarchy",
"-fh",
"[none|all|except <List Strings>|only <List Strings>|leaf]",
"Specifies how the user hierarchy will be flattened. [none] means",
"the hierarchy is preserved and [all] that it is completely flattened.",
"[except] followed by a list of String specifies which module will",
"not be flattened and [only] has the opposite meaning. [leaf] means",
"that only the leaves modules will be flattened. Default is [none].",
GnlFlattenOption,
},
{"-flat_string",
"-fs",
"<String>",
"Specifies the String which separates instance names on a ",
"hierarchical path after flattening. By default the string is \".\\\".",
"Ex: U1.U2.U3 means that after the flattening, the current component",
"comes from the instance path U1, U2 and U3.",
NULL,
GnlFlattenStrOption,
GNL_OPTION_FILE
},
{"-flat_instance_name",
"-fin",
"[hierarchical|compact]",
"Specifies if the names of the flattened instances will correspond",
"to the hierarchical name (ex: \\inst1\\inst2) or a compact name (name",
"of the deepest instance) when flattening of the hierarchy is invoked.",
NULL,
NULL,
GnlFlattenInstanceNameOption,
GNL_OPTION_FILE
},
{"-no_fold_identical_partition",
"-nfip",
"",
"Specifies if during the optimization phase, HUBBLE-RTL will try to fold",
"or not the identical partition of glue logic. This can significantly",
"speed-up the logic optimization phase. By default the folding is",
"performed but by using this option you disable the folding.",
NULL,
GnlFoldIsoPartOption,
GNL_OPTION_FILE
}

```

```

},
{"-top",
"-t",
"<String>",
"Specifies the top Verilog module from which the synthesis/estimation",
"will be performed.",
NULL,
NULL,
NULL,
GnlTopOption,
GNL_OPTION_FILE
},
{"-y",
"-dir",
"<String>",
"Specifies the directory where leaf cell modules can be found.",
NULL,
NULL,
NULL,
NULL,
GnlModelDirOption,
GNL_OPTION_FILE
},
{"+libext",
"-ext",
"<String>",
"Specifies the extension of the files defining the leaf cell modules.",
NULL,
NULL,
NULL,
NULL,
GnlModelExtOption,
GNL_OPTION_FILE
},
{"-build_lib_force",
"-blf",
"[min|max]",
"Specifies if the .lib library will be used in a minimum or a",
"maximum use. Maximum use is slower to build the library but",
"results may be better because the complex cells may be instantiated.",
NULL,
NULL,
GnlBuildLibForceOption,
GNL_OPTION_FILE
},
{"-print_hierarchy",
"-ph",
"[none|before_flat|after_flat|both]",
"Invokes the printing of the design hierarchy. This can be done",
"before the flattening phase (option [before_flat]) or after it",
"(option [after_flat]) since the flattening can change the hierarchy.",
"Option [both] prints out before and after flattening.",
NULL,
GnlPrintHierarchyOption,
GNL_OPTION_FILE
},
{"-report_data",
}

```

gnloption.c

```
--rd",
" [none|modules|cells|timings|powers]",
"Invokes the printing of a final report in which several data are",
"presented related either to modules (use option [modules]) or to",
"library cells (use option [cells]). Use option [timings] to get",
"any information regarding timing issues in the design. Option",
"[powers] will give average power and peak power in the design.",
GnlReportDataOption,
GNL_OPTION_FILE
},
{"-print_nb_critical_path",
"-pncp",
"<Integer>",
"Defines the number of critical paths which will be printed out",
"when the option [-rd timings] is invoked",
NULL,
NULL,
NULL,
GnlPrintNbCritPath,
GNL_OPTION_FILE
},
{"-print_max_fanout_vars",
"-pmfv",
"",
"when the option [-rd cells] is invoked, this option forces HUBBLE-RTL",
"to print out for each library cell, the corresponding instances",
"with the maximum fan out",
NULL,
NULL,
NULL,
GnlPrintMaxFanoutVarsOption,
GNL_OPTION_FILE
},
/* For VERIFICATION */
{"-input1",
"-i1",
"<String>",
"Specifies the first Netlist used in the Netlist Checker.",
NULL,
NULL,
NULL,
NULL,
GnlInputOption,
GNL_OPTION_FILE
},
{"-input2",
"-i2",
"<String>",
"Specifies the second Netlist used in the Netlist Checker.",
NULL,
NULL,
NULL,
NULL,
GnlReferenceOption,
GNL_OPTION_FILE
},
{"-read_name_association_file",
```

gnloption.c

```
"-rnaf",
"<String>",
"Specifies the input file which defines the associations between the",
"port names. The Netlist Checker will take into account these",
"associations to prove the two netlists. If this option is not",
"invoked then HUBBLE will try to figure out the associations by",
"itself.",
GnlInputAssocOption,
GNL_OPTION_FILE
},
{"-print_name_association_file",
"-pnaf",
"<String>",
"Tells HUBBLE to print out the file of the ports associations used",
"in the Netlist Checker. This file can be edited and re-read in HUBBLE",
"with the option '-naif'. The ports associations can be then controlled",
"in that way if you are not satisfied with the automatic ports",
"association done in HUBBLE.",
GnlOutputAssocOption,
GNL_OPTION_FILE
},
{"-verif_max_bdd_node",
"-vmaxbdd",
"<Integer>",
"Specifies the maximum number of BDDs node before any break point",
"is introduced when the Equivalence Checker is invoked",
NULL,
NULL,
NULL,
GnlMaxBddNodeOption,
GNL_OPTION_FILE
},
{"-ignore_random_simulation",
"-irs",
"",
"tells HUBBLE not to call the random simulation phase during the Netlist",
"Checking.",
NULL,
NULL,
NULL,
GnlIgnoreRandomSimOption,
GNL_OPTION_FILE
}

};

/* -----
 * Global Variables.
 * -----
 GNL_ENV      G_GnlEnv;
int         G_GLOB_ARGC;
char        **G_GLOB_argv;

/* ----- */
```

gnloption.c

```
/* GnlSetDefaultEnv */  
/* ----- */  
/* Set default options on global environment 'G_GnlEnv'. */  
/* ----- */  
GNL_STATUS GnlSetDefaultEnv ()  
{  
    char *NewStr;  
  
    /* -1 corresponds to the default injection. */  
    SetGnlEnvInject (-1);  
  
    /* we prints out by default 1 critical path. */  
    SetGnlEnvPrintNbCritPath (1);  
  
    /* we perform by default an automatic folding of logic partitions */  
    SetGnlEnvFoldIsoPart (1);  
  
    /* The separation Str for flattening is '.' */  
    if (GnlStrCopy (".", &NewStr))  
        return (GNL_MEMORY_FULL);  
    SetGnlEnvFlattenStr (NewStr);  
  
    /* By default we try to respect the tech. library constraints */  
    SetGnlEnvRespectLibConstraints (1);  
  
    /* By default we invoke the post optimization phase. */  
    SetGnlEnvPostOpt (1);  
  
    /* By default the voltage is 5.0 */  
    SetGnlEnvVDD (5.0);  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* GnlCreateGnlEnv */  
/* ----- */  
/* Creates and set global environment 'G_GnlEnv'. */  
/* ----- */  
GNL_STATUS GnlCreateGnlEnv ()  
{  
  
    if ((G_GnlEnv = (GNL_ENV)malloc (1, sizeof (GNL_ENV_REC))) == NULL)  
        return (GNL_MEMORY_FULL);  
  
    /* Setting the default values for the Environment 'Env'. */  
    if (GnlSetDefaultEnv ())  
        return (GNL_MEMORY_FULL);  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* GnlPrintHelp */  
/* ----- */  
void GnlPrintHelp ()  
{
```

gnloption.c

```
int opt_no;

fprintf (stderr, "\nUsage: %s %s\n", GNL_TOOL_NAME, GNL_MAPIT_VERSION);
fprintf (stderr, "-----\n");
for (opt_no=0; opt_no < GNL_NB_OPTIONS; opt_no++)
{
    fprintf (stderr, " %s [%s] %s : \n",
            G_Options[opt_no].Name,
            G_Options[opt_no].Alias,
            G_Options[opt_no].OptionValue);
    if (G_Options[opt_no].Help1)
        fprintf (stderr, "    %s \n", G_Options[opt_no].Help1);
    if (G_Options[opt_no].Help2)
        fprintf (stderr, "    %s \n", G_Options[opt_no].Help2);
    if (G_Options[opt_no].Help3)
        fprintf (stderr, "    %s \n", G_Options[opt_no].Help3);
    if (G_Options[opt_no].Help4)
        fprintf (stderr, "    %s \n", G_Options[opt_no].Help4);
    if (G_Options[opt_no].Help5)
        fprintf (stderr, "    %s \n", G_Options[opt_no].Help5);
    fprintf (stderr, "\n");
}

/*
 * -----
 * GnlParamIsOption                                */
/* -----
 * Returns 1 if 'Param' is a Mapit option and 0 otherwise.      */
/* ----- */

int GnlParamIsOption (Param)
char *Param;
{
    int opt_no;

    for (opt_no=0; opt_no < GNL_NB_OPTIONS; opt_no++)
    {
        if (!strcmp (Param, G_Options[opt_no].Name))
            return (1);
        if (!strcmp (Param, G_Options[opt_no].Alias))
            return (1);
    }
    return (0);
}

/*
 * -----
 * GnlParseCommandLine                                */
/* -----
 * WARNING: when processing option -command, global variables      */
/* G_GLOB_ARGC and G_GLOB_argv can be modified (when function      */
/* "GnlReadCmdFile" is called).                                */
/* ----- */

void GnlParseCommandLine (GnlEnv, argc, argv)
    GNL_ENV GnlEnv;
```

gnloption.c

```
int          argc;
char        **argv;
{
    int          arg_no = 1;
    int          opt_no;
    int          i;
    int          j;
    BLIST      *SubList;
    int          Found;

    if (argc == 1)
    {
        GnlPrintHelp ();
        exit (0);
    }

    G_GLOB_ARGC = argc;
    G_GLOB_ARGV = argv;

    while (arg_no < G_GLOB_ARGC)
    {
        Found = 0;
        for (opt_no=0; opt_no < GNL_NB_OPTIONS; opt_no++)
        {
            if (!strcmp (G_GLOB_ARGV[arg_no],
                         G_Options[opt_no].Name))
            {
                Found = 1;
                break;
            }
        }

        if (!Found)
        {
            for (opt_no=0; opt_no < GNL_NB_OPTIONS; opt_no++)
            {
                if (!strcmp (G_GLOB_ARGV[arg_no],
                             G_Options[opt_no].Alias))
                {
                    Found = 1;
                    break;
                }
            }
        }

        if (!Found)
        {
            fprintf (stderr, " ERROR: option [%s] is unknown\n",
                     G_GLOB_ARGV[arg_no]);
            exit (1);
        }

        (G_Options[opt_no].Function) (GnlEnv, G_GLOB_ARGC,
                                     G_GLOB_ARGV,
                                     &arg_no);
    }
}
```

gnloption.c

```
/* ----- */  
/* GnlCheckOptionsSynthesis */  
/* ----- */  
/* For synthesis purpose we make sure that: */  
/* - there is an input file */  
/* - there is an output file */  
/* - there is a technology library provided */  
/* ----- */  
void GnlCheckOptionsSynthesis ()  
{  
  
    if (!GnlEnvInput())  
    {  
        fprintf (stderr,  
            " ERROR: Verilog input file is missing (use -i option)\n");  
        exit (1);  
    }  
  
    if (!GnlEnvOutput())  
    {  
        fprintf (stderr,  
            " ERROR: Verilog output file name is missing (use -o option)\n");  
        exit (1);  
    }  
  
    if (!GnlEnvLib())  
    {  
        fprintf (stderr,  
            " ERROR: Technology Library must be specified (use -lib option)\n");  
        exit (1);  
    }  
}  
  
/* ----- */  
/* GnlCheckOptionsEstimation */  
/* ----- */  
/* For estimation purpose we make sure that: */  
/* - there is an input file */  
/* - there is a technology library provided */  
/* - the input format is VERILOG */  
/* ----- */  
void GnlCheckOptionsEstimation ()  
{  
  
    if (!GnlEnvInput())  
    {  
        fprintf (stderr,  
            " ERROR: Verilog input file is missing (use -i option)\n");  
        exit (1);  
    }  
  
    if (!GnlEnvLib())  
    {  
        fprintf (stderr,  
            " ERROR: Technology Library must be specified (use -lib option)\n");  
        exit (1);  
    }  
}
```

```

}

if (GnlEnvInputFormat() != GNL_INPUT_VLG)
{
    fprintf (stderr,
    " ERROR: The input format must be verilog in ESTIMATION mode\n");
    exit (1);
}

/*
/* GnlCheckOptionsPostOptimization
/*
/*
/* For estimation purpose we make sure that:
/*           - there is an input file
/*           - there is an output file
/*           - there is a technology library provided
/*           - the input format is VERILOG
/*
void GnlCheckOptionsPostOptimization ()
{

    if (!GnlEnvInput())
    {
        fprintf (stderr,
        " ERROR: Verilog input file is missing (use -i option)\n");
        exit (1);
    }

    if (!GnlEnvLib())
    {
        fprintf (stderr,
    " ERROR: Technology Library must be specified (use -lib option)\n");
        exit (1);
    }

    if (GnlEnvInputFormat() != GNL_INPUT_VLG)
    {
        fprintf (stderr,
        " ERROR: The input format must be verilog in POST-OPTIMIZATION mode\n");
        exit (1);
    }

    if (!GnlEnvOutput())
    {
        fprintf (stderr,
        " ERROR: Verilog output file name is missing (use -o option)\n");
        exit (1);
    }
}

/*
/* GnlCheckOptionsTranslate
/*
/*
/* For translate purpose we make sure that:
/*           - there is an input file
/*

```

gnloption.c

```
/* ----- */
void GnlCheckOptionsTranslate ()
{
    if (!GnlEnvInput())
    {
        fprintf (stderr,
            " ERROR: fsm input file is missing (use -i option)\n");
        exit (1);
    }
}

/* ----- */
/* GnlCheckOptionsVerification */
/* ----- */
/* For verification purpose we make sure that:
   - there is a reference file
   - there is an input file
   - there is a technology library provided
*/
void GnlCheckOptionsVerification ()
{
    if (!GnlEnvInput())
    {
        fprintf (stderr,
            " ERROR: Verilog input file is missing (use -i option)\n");
        exit (1);
    }

    if (!GnlEnvReference())
    {
        fprintf (stderr,
            " ERROR: Reference Verilog Netlist is missing (use -r option)\n");
        exit (1);
    }

    if (!GnlEnvLib())
    {
        fprintf (stderr,
            " ERROR: Technology Library must be specified (use -lib option)\n");
        exit (1);
    }
}

/* ----- */
/* GnlCheckOptions */
/* ----- */
void GnlCheckOptions (Env)
    GNL_ENV      Env;
{
    switch (GnlEnvMode ()) {
        case GNL_MODE_SYNTHESIS:
            GnlCheckOptionsSynthesis ();
            break;
    }
}
```

gnloption.c

```
case GNL_MODE_ESTIMATION:
    GnlCheckOptionsEstimation ();
    break;

case GNL_MODE_TRANSLATE:
    GnlCheckOptionsTranslate ();
    break;

case GNL_MODE_VERIFICATION:
    GnlCheckOptionsVerification ();
    break;

case GNL_MODE_POST_OPTIMIZATION:
    GnlCheckOptionsPostOptimization ();
    break;
}

}

/* -----
/* GnlModeOption
/* ----- */
void GnlModeOption (Env, argc, argv, arg_no)
    GNL_ENV Env;
    int argc;
    char **argv;
    int *arg_no;
{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "estimation") &&
        strcmp (argv[*arg_no], "verification") &&
        strcmp (argv[*arg_no], "translate") &&
        strcmp (argv[*arg_no], "post_optimization") &&
        strcmp (argv[*arg_no], "synthesis"))
    {
        fprintf (stderr,
        " ERROR: Option values are
[synthesis|estimation|translate|verification|post_optimization]\n");
        exit (1);
    }

    if (!strcmp (argv[*arg_no], "estimation"))
        SetGnlEnvMode (GNL_MODE_ESTIMATION);
    else if (!strcmp (argv[*arg_no], "verification"))
        SetGnlEnvMode (GNL_MODE_VERIFICATION);
    else if (!strcmp (argv[*arg_no], "translate"))
        SetGnlEnvMode (GNL_MODE_TRANSLATE);
    else if (!strcmp (argv[*arg_no], "post_optimization"))
        SetGnlEnvMode (GNL_MODE_POST_OPTIMIZATION);
```

gnloption.c

```
else
    SetGnlEnvMode (GNL_MODE_SYNTHESIS);
    (*arg_no)++;
}

/*
/* GnlInputOption
*/
void GnlInputOption (Env, argc, argv, arg_no)
    GNL_ENV Env;
    int      argc;
    char    **argv;
    int      *arg_no;
{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
            argv[--(*arg_no)]);
        exit (1);
    }

    /* If the input file name has the same name as the output file name */
    if (GnlEnvOutput() && !strcmp (GnlEnvOutput(), argv[*arg_no]))
    {
        fprintf (stderr,
            " ERROR: input and output file cannot be the same\n");
        exit (1);
    }

    SetGnlEnvInput (argv[*arg_no]);
    (*arg_no)++;
}

/*
/* GnlInputAssocOption
*/
void GnlInputAssocOption (Env, argc, argv, arg_no)
    GNL_ENV Env;
    int      argc;
    char    **argv;
    int      *arg_no;
{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
            argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvInputAssoc (argv[*arg_no]);
    (*arg_no)++;
}

/*
/* GnlOutputOption
*/
```

gnloption.c

```
/* -----
void GnlOutputOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char      **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                argv[--(*arg_no)]);
        exit (1);
    }

    /* If the input file name has the same name as the output file name */
    if (GnlEnvInput() && !strcmp (GnlEnvInput(), argv[*arg_no]))
    {
        fprintf (stderr,
                " ERROR: input and output file cannot be the same\n");
        exit (1);
    }

    SetGnlEnvOutput (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlOutputAssocOption
/* -----
void GnlOutputAssocOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char      **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvOutputAssoc (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlXorUnprovedOption
/* -----
void GnlXorUnprovedOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char      **argv;
```

gnloption.c

```
int          *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvXorUnproved (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlLogOption
/* -----
void GnlLogOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    fprintf (stderr, "Mapit Trace redirected to log file '%s'\n",
             argv[*arg_no]);
    fprintf (stderr, "Mapit Processing..\n");

    if (!freopen(argv[*arg_no] , "w", stderr))
    {
        fprintf (stderr, "ERROR: cannot redirect <stderr>.\n");
        exit (1);
    }

    SetGnlEnvLog (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlLibraryOption
/* -----
void GnlLibraryOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;
```

gnloption.c

```
{  
    (*arg_no)++;  
    if (argc == *arg_no)  
    {  
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);  
        exit (1);  
    }  
  
    setGnlEnvLib (argv[*arg_no]);  
    (*arg_no)++;  
}  
  
/* ----- */  
/* GnlInputFormatOption */  
/* ----- */  
void GnlInputFormatOption (Env, argc, argv, arg_no)  
{  
    GNL_ENV      Env;  
    int         argc;  
    char      **argv;  
    int         *arg_no;  
  
{  
    (*arg_no)++;  
    if (argc == *arg_no)  
    {  
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[-(*arg_no)]);  
        exit (1);  
    }  
  
    if (strcmp (argv[*arg_no], "fsm") &&
        strcmp (argv[*arg_no], "vlg"))  
    {  
        fprintf (stderr,
                  " ERROR: Option values are [fsm|vlg]\n");
        exit (1);  
    }  
  
    if (!strcmp (argv[*arg_no], "fsm"))
        SetGnlEnvInputFormat (GNL_INPUT_FSM);
    else
        SetGnlEnvInputFormat (GNL_INPUT_VLG);  
  
    (*arg_no)++;  
}  
  
/* ----- */  
/* GnlFlattenStrOption */  
/* ----- */  
void GnlFlattenStrOption (Env, argc, argv, arg_no)  
{  
    GNL_ENV      Env;  
    int         argc;  
    char      **argv;  
    int         *arg_no;
```

gnloption.c

```
BLIST      ListFlatten;
char       *Param;

(*arg_no)++;
if (argc == *arg_no)
{
    fprintf (stderr, " ERROR: value for option [%s] is missing\n",
            argv[--(*arg_no)]);
    exit (1);
}

SetGnlEnvFlattenStr (argv[*arg_no]);

(*arg_no)++;
}

/* -----
/* GnlFlattenOption
/* -----
void GnlFlattenOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int          argc;
    char        **argv;
    int          *arg_no;

{
    BLIST      ListFlatten;
    char       *Param;

(*arg_no)++;
if (argc == *arg_no)
{
    fprintf (stderr, " ERROR: value for option [%s] is missing\n",
            argv[--(*arg_no)]);
    exit (1);
}

if (strcmp (argv[*arg_no], "none") &&
    strcmp (argv[*arg_no], "leaf") &&
    strcmp (argv[*arg_no], "only") &&
    strcmp (argv[*arg_no], "except") &&
    strcmp (argv[*arg_no], "all"))
{
    fprintf (stderr,
            " ERROR: Option values are [none|all|except|onlyleaf]\n");
    exit (1);
}

if (!strcmp (argv[*arg_no], "none"))
    SetGnlEnvFlatten (GNL_NO_FLATTEN);
else if (!strcmp (argv[*arg_no], "except"))
    SetGnlEnvFlatten (GNL_EXCEPT_FLATTEN);
else if (!strcmp (argv[*arg_no], "only"))
    SetGnlEnvFlatten (GNL_ONLY_FLATTEN);
else if (!strcmp (argv[*arg_no], "leaf"))
```

gnloption.c

```
SetGnlEnvFlatten (GNL_LEAF_FLATTEN);
else
    SetGnlEnvFlatten (GNL_ALL_FLATTEN);

(*arg_no)++;

if ((GnlEnvFlatten () == GNL_NO_FLATTEN) ||
    (GnlEnvFlatten () == GNL_LEAF_FLATTEN) ||
    (GnlEnvFlatten () == GNL_ALL_FLATTEN))
    return;

if (BListCreate (&ListFlatten))
    return;
SetGnlEnvFlatListModules (ListFlatten);

while (1)
{
    if (argc == *arg_no)
        return;
    Param = argv[*arg_no];

    if (GnlParamIsOption (Param))
    {
        return;
    }

    if (BListAddElt (ListFlatten, (int)Param))
        return;

    (*arg_no)++;
}

/*
/* GnlFlattenInstanceNameOption
*/
void GnlFlattenInstanceNameOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "hierarchical") &&
        strcmp (argv[*arg_no], "compact"))
    {
        fprintf (stderr,
                 " ERROR: Option values are [hierarchical|compact]\n");
    }
}
```

gnloption.c

```
    exit (1);
}

if (!strcmp (argv[*arg_no], "hierarchical"))
    SetGnlEnvFlattenInstanceName (GNL_FLAT_HIERARCHICAL_NAME);
else
    SetGnlEnvFlattenInstanceName (GNL_FLAT_COMPACT_NAME);

(*arg_no)++;
}

/* -----
/* GnlInjectOption
/* -----
void GnlInjectOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvInject (atoi (argv[*arg_no]));
    (*arg_no)++;
}

/* -----
/* GnlOptForceOption
/* -----
void GnlOptForceOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "low") &&
        strcmp (argv[*arg_no], "medium") &&
        strcmp (argv[*arg_no], "high"))
    {
        fprintf (stderr,
                 " ERROR: Option values are [low|medium|high]\n");
    }
}
```

gnloption.c

```
    exit (1);
}

if (!strcmp (argv[*arg_no], "low_opt"))
    SetGnlEnvOptForce (GNL_LOW_OPT);
else if (!strcmp (argv[*arg_no], "med_opt"))
    SetGnlEnvOptForce (GNL_MED_OPT);
else
    SetGnlEnvOptForce (GNL_HIGH_OPT);

(*arg_no)++;
}

/* -----
/* GnlPrintHierarchyOption
/* -----
void GnlPrintHierarchyOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "none") &&
        strcmp (argv[*arg_no], "before_flat") &&
        strcmp (argv[*arg_no], "both") &&
        strcmp (argv[*arg_no], "after_flat"))
    {
        fprintf (stderr,
                 " ERROR: Option values are
[none|before_flat|after_flat|both]\n");
        exit (1);
    }

    if (!strcmp (argv[*arg_no], "none"))
        SetGnlEnvPrintHierarchy (GNL_PRINT_NONE);
    else if (!strcmp (argv[*arg_no], "before_flat"))
        SetGnlEnvPrintHierarchy (GNL_PRINT_BEFORE_FLAT);
    else if (!strcmp (argv[*arg_no], "both"))
        SetGnlEnvPrintHierarchy (GNL_PRINT_BOTH);
    else
        SetGnlEnvPrintHierarchy (GNL_PRINT_AFTER_FLAT);

    (*arg_no)++;
}

/* -----
/* GnlReportDataOption
/* ----- */
```

gnloption.c

```
void GnlReportDataOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "none") &&
        strcmp (argv[*arg_no], "modules") &&
        strcmp (argv[*arg_no], "timings") &&
        strcmp (argv[*arg_no], "powers") &&
        strcmp (argv[*arg_no], "cells"))
    {
        fprintf (stderr,
                 " ERROR: Option values are
[none|modules|cells|timings|powers]\n");
        exit (1);
    }

    if (!strcmp (argv[*arg_no], "none"))
        SetGnlEnvReportData (GNL_REPORT_NONE);
    else if (!strcmp (argv[*arg_no], "modules"))
        SetGnlEnvReportData (GNL_REPORT_MODULES);
    else if (!strcmp (argv[*arg_no], "timings"))
        SetGnlEnvReportData (GNL_REPORT_TIMINGS);
    else if (!strcmp (argv[*arg_no], "powers"))
        SetGnlEnvReportData (GNL_REPORT POWERS);
    else
        SetGnlEnvReportData (GNL_REPORT CELLS);

    (*arg_no)++;
}

/*
 * -----
 * GnlCriterionOption
 * -----
 */
void GnlCriterionOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }
```

gnloption.c

```
}

if (strcmp (argv[*arg_no], "area_gate") &&
    strcmp (argv[*arg_no], "power") &&
    strcmp (argv[*arg_no], "area_net") &&
    strcmp (argv[*arg_no], "nl_delay") &&
    strcmp (argv[*arg_no], "timing"))
{
    fprintf (stderr,
        " ERROR: Option values are
[area_gate|area_net|nl_delay|power|timing]\n");
    exit (1);
}

if (!strcmp (argv[*arg_no], "area_gate"))
    SetGnlEnvCriterion (GNL_AREA);
else if (!strcmp (argv[*arg_no], "power"))
    SetGnlEnvCriterion (GNL_POWER);
else if (!strcmp (argv[*arg_no], "nl_delay"))
    SetGnlEnvCriterion (GNL_NL_DELAY);
else if (!strcmp (argv[*arg_no], "area_net"))
    SetGnlEnvCriterion (GNL_NB_NETS);
else
    SetGnlEnvCriterion (GNL_TIMING);

(*arg_no)++;
}

/* -----
/* GnlTopOption
/* -----
void GnlTopOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
            argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvTop (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlModelDirOption
/* -----
void GnlModelDirOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
```

gnloption.c

```
int          *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvModelDir (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlFsmDirOption
/* -----
void GnlFsmDirOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvFsmDir (argv[*arg_no]);
    (*arg_no)++;
}

/* -----
/* GnlModelExtOption
/* -----
void GnlModelExtOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvModelExt (argv[*arg_no]);
    (*arg_no)++;
}
```

gnloption.c

```
}

/* -----
/* GnlBuildLibForceOption                               */
/* ----- */
void GnlBuildLibForceOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                  argv[--(*arg_no)]);
        exit (1);
    }

    if (strcmp (argv[*arg_no], "min") &&
        strcmp (argv[*arg_no], "max"))
    {
        fprintf (stderr,
                  " ERROR: Option values are [min|max]\n");
        exit (1);
    }

    if (!strcmp (argv[*arg_no], "min"))
        SetGnlEnvBuildLibForce (GNL_BUILD_LIB_MIN);
    else
        SetGnlEnvBuildLibForce (GNL_BUILD_LIB_MAX);
    (*arg_no)++;
}

/* -----
/* GnlPrintMaxFanoutVarsOption                         */
/* ----- */
void GnlPrintMaxFanoutVarsOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvPrintMaxFanoutVars (1);
    (*arg_no)++;
}

/* -----
/* GnlFoldIsoPartOption                               */
/* ----- */
void GnlFoldIsoPartOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;
```

gnloption.c

```
    {
        SetGnlEnvFoldIsoPart (0);
        (*arg_no)++;
    }

/* -----
/* GnlUseQBarOption
/* -----
void GnlUseQBarOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    SetGnlEnvUseQBar (1);
    (*arg_no)++;
}

/* -----
/* GnlReplaceOnlyCombiCompoOption
/* -----
void GnlReplaceOnlyCombiCompoOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    SetGnlEnvReplaceOnlyCombiCompo (1);
    (*arg_no)++;
}

/* -----
/* GnlDontTouchOption
/* -----
void GnlDontTouchOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;

{
    SetGnlEnvDontTouch (1);
    (*arg_no)++;
}

/* -----
/* GnlRemoveBufferOption
/* -----
void GnlRemoveBufferOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char       **argv;
    int         *arg_no;
```

gnloption.c

```
{  
    SetGnlEnvRemoveBuffer (1);  
    (*arg_no)++;  
}  
  
/* ----- */  
/* GnlPinSeqPolMatchingOption */  
/* ----- */  
void GnlPinSeqPolMatchingOption (Env, argc, argv, arg_no)  
    GNL_ENV      Env;  
    int         argc;  
    char        **argv;  
    int         *arg_no;  
  
{  
    SetGnlEnvPinSeqPolMatching (1);  
    (*arg_no)++;  
}  
  
/* ----- */  
/* GnlMaxCellSizeSupportOption */  
/* ----- */  
void GnlMaxCellSizeSupportOption (Env, argc, argv, arg_no)  
    GNL_ENV      Env;  
    int         argc;  
    char        **argv;  
    int         *arg_no;  
  
{  
    (*arg_no)++;  
    if (argc == *arg_no)  
    {  
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                  argv[--(*arg_no)]);  
        exit (1);  
    }  
  
    SetGnlEnvMaxCellSizeSupport (atoi (argv[*arg_no]));  
    (*arg_no)++;  
}  
  
/* ----- */  
/* GnlMinShareLogicSizeOption */  
/* ----- */  
void GnlMinShareLogicSizeOption (Env, argc, argv, arg_no)  
    GNL_ENV      Env;  
    int         argc;  
    char        **argv;  
    int         *arg_no;  
  
{  
    (*arg_no)++;  
    if (argc == *arg_no)  
    {  
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                  argv[--(*arg_no)]);  
        exit (1);  
    }
```

gnloption.c

```
}

SetGnlEnvMinShareLogicSize (atoi (argv[*arg_no]));
(*arg_no)++;
}

/* -----
/* GnlVDDOption
/* -----
void GnlVDDOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    float      Volt;

    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    Volt = (float)atoi (argv[*arg_no])/(float)1000;

    SetGnlEnvVDD (Volt);

    (*arg_no)++;
}

/* -----
/* GnlRespectLibConstraintsOption
/* -----
void GnlRespectLibConstraintsOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvRespectLibConstraints (0);
    (*arg_no)++;
}

/* -----
/* GnlIgnorePostOptOption
/* -----
void GnlIgnorePostOptOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
```

gnloption.c

```
SetGnlEnvPostOpt (0);
(*arg_no)++;
}

/* -----
/* GnlPrintResizedGatesOption
/* -----
void GnlPrintResizedGatesOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvPrintResizedGates (1);
    (*arg_no)++;
}

/* -----
/* GnlIgnoreRandomSimOption
/* -----
void GnlIgnoreRandomSimOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvIgnoreRandomSim (1);
    (*arg_no)++;
}

/* -----
/* GnlUseVerilogPrimitivesOption
/* -----
void GnlUseVerilogPrimitivesOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvUseVerilogPrimitives (1);
    (*arg_no)++;
}

/* -----
/* GnlPrintClkDomainOption
/* -----
void GnlPrintClkDomainOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvPrintClkDomain (1);
```

gnloption.c

```
    (*arg_no)++;
}

/* -----
/* GnlPrintCritRegionOption
/* -----
void GnlPrintCritRegionOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    SetGnlEnvPrintCritRegion (1);
    (*arg_no)++;
}

/* -----
/* GnlPrintNbCritPath
/* -----
void GnlPrintNbCritPath (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int         argc;
    char        **argv;
    int         *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvPrintNbCritPath (atoi (argv[*arg_no]));
    (*arg_no)++;
}

/* -----
/* GnlReferenceOption
/* -----
void GnlReferenceOption (Env, argc, argv, arg_no)
    GNL_ENV  Env;
    int      argc;
    char    **argv;
    int      *arg_no;
{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvReference (argv[*arg_no]);
}
```

gnloption.c

```

        (*arg_no)++;
    }

/* -----
/* GnlMaxBddNodeOption
/* -----
void GnlMaxBddNodeOption (Env, argc, argv, arg_no)
    GNL_ENV      Env;
    int          argc;
    char        **argv;
    int          *arg_no;

{
    (*arg_no)++;
    if (argc == *arg_no)
    {
        fprintf (stderr, " ERROR: value for option [%s] is missing\n",
                 argv[--(*arg_no)]);
        exit (1);
    }

    SetGnlEnvMaxBddNode (atoi (argv[*arg_no]));
    (*arg_no)++;
}

/* -----

```

gnloption.h

gnloption.h

```
typedef enum {
    GNL_NO_FLATTEN,
    GNL_EXCEPT_FLATTEN,
    GNL_ONLY_FLATTEN,
    GNL_LEAF_FLATTEN,
    GNL_ALL_FLATTEN
}
    GNL_FLATTEN;

/* -----
/* GNL_FLATTEN_INSTANCE_NAME
/* -----
typedef enum {
    GNL_FLAT_HIERARCHICAL_NAME,
    GNL_FLAT_COMPACT_NAME
}
    GNL_FLATTEN_INSTANCE_NAME;

/* -----
/* GNL_BUILD_LIB_FORCE
/* -----
typedef enum {
    GNL_BUILD_LIB_MIN,
    GNL_BUILD_LIB_MAX
}
    GNL_BUILD_LIB_FORCE;

/* -----
/* GNL_INPUT_FORMAT
/* -----
typedef enum {
    GNL_INPUT_FSM,
    GNL_INPUT_VLG
}
    GNL_INPUT_FORMAT;

/* -----
/* GNL_PRINT_HIERARCHY
/* -----
typedef enum {
    GNL_PRINT_NONE,
    GNL_PRINT_BEFORE_FLAT,
    GNL_PRINT_AFTER_FLAT,
    GNL_PRINT_BOTH
}
    GNL_PRINT_HIERARCHY;

/* -----
/* GNL_REPORT_DATA
/* -----
typedef enum {
    GNL_REPORT_NONE,
    GNL_REPORT_MODULES,
    GNL_REPORT_TIMINGS,
    GNL_REPORT_POWERS,
    GNL_REPORT_CELLS
}
```

gnloption.h

```
GNL_REPORT_DATA;

/* ----- */
/* GNL_ENV */
/* ----- */
/* Global environment structure in MAPIT which stores all the options */
/* which can be relevant during any process. */
/* ----- */
/*----- */
typedef struct
{
    GNL_MODE           Mode;
    char               *Input;
    GNL_INPUT_FORMAT   InputFormat;
    char               *Output;
    char               *Log;
    GNL_OPT_FORCE     OptForce;
    GNL_CRITERION     Criterion;
    int                UseQBar;
    int                PinSeqPolMatching;
    int                Inject;
    GNL_FLATTEN        Flatten;
    char               *FlattenStr;
    BLIST              FlatListModules;
    GNL_FLATTEN_INSTANCE_NAME FlattenInstanceName;
    char               *Lib;
    char               *Top;
    char               *ModelDir;
    char               *ModelExt;
    GNL_BUILD_LIB_FORCE BuildLibForce;
    char               *FsmDir;
    GNL_PRINT_HIERARCHY PrintHierarchy;
    GNL_REPORT_DATA    ReportData;
    int                PrintNbCritPath;
    int                FoldIsoPart;
    int                PrintMaxFanoutVars;
    int                MaxCellSizeSupport;
    int                MinShareLogicSize;
    int                UseVerilogPrimitives;
    float              VDD;
    int                ReplaceOnlyCombiCompo;
    int                RemoveBuffer;
    int                DontTouch;

    /* Options For VERIFICATION */
    char               *Reference; /* VERIFICATION */
    int                MaxBddNode; /* VERIFICATION */
    char               *InputAssoc;
    char               *OutputAssoc;
    char               *XorUnproved;
    int                IgnoreRandomSim;

    /* Options for TIMING reports */
    int                RespectLibConstraints;
    int                PrintClkDomain;
    int                PrintCritRegion;
    int                DoPostOpt;
}
```

```

    int                               PrintResizedGates;
}

GNL_ENV_REC, *GNL_ENV;

/* We access the global variable 'G_GnlEnv'. */
#define GnlEnvMode()                  (G_GnlEnv->Mode)
#define GnlEnvInput()                 (G_GnlEnv->Input)
#define GnlEnvInputFormat()            (G_GnlEnv->InputFormat)
#define GnlEnvOutput()                (G_GnlEnv->Output)
#define GnlEnvLog()                   (G_GnlEnv->Log)
#define GnlEnvOptForce()              (G_GnlEnv->OptForce)
#define GnlEnvCriterion()             (G_GnlEnv->Criterion)
#define GnlEnvUseQBar()               (G_GnlEnv->UseQBar)
#define GnlEnvPinSeqPolMatching()     (G_GnlEnv->PinSeqPolMatching)
#define GnlEnvInject()                (G_GnlEnv->Inject)
#define GnlEnvFlatten()               (G_GnlEnv->Flatten)
#define GnlEnvFlattenStr()             (G_GnlEnv->FlattenStr)
#define GnlEnvFlatListModules()        (G_GnlEnv->FlatListModules)
#define GnlEnvFlattenInstanceName()    (G_GnlEnv->FlattenInstanceName)
#define GnlEnvLib()                   (G_GnlEnv->Lib)
#define GnlEnvTop()                   (G_GnlEnv->Top)
#define GnlEnvModelDir()              (G_GnlEnv->ModelDir)
#define GnlEnvModelExt()              (G_GnlEnv->ModelExt)
#define GnlEnvBuildLibForce()          (G_GnlEnv->BuildLibForce)
#define GnlEnvFsmDir()                (G_GnlEnv->FsmDir)
#define GnlEnvPrintHierarchy()         (G_GnlEnv->PrintHierarchy)
#define GnlEnvReportData()             (G_GnlEnv->ReportData)
#define GnlEnvPrintNbCritPath()        (G_GnlEnv->PrintNbCritPath)
#define GnlEnvFoldIsoPart()            (G_GnlEnv->FoldIsoPart)
#define GnlEnvPrintMaxFanoutVars()     (G_GnlEnv->PrintMaxFanoutVars)
#define GnlEnvRespectLibConstraints()  (G_GnlEnv->RespectLibConstraints)
#define GnlEnvPrintClkDomain()          (G_GnlEnv->PrintClkDomain)
#define GnlEnvPrintCritRegion()         (G_GnlEnv->PrintCritRegion)
#define GnlEnvMaxCellSizeSupport()      (G_GnlEnv->MaxCellSizeSupport)
#define GnlEnvMinShareLogicSize()       (G_GnlEnv->MinShareLogicSize)
#define GnlEnvUseVerilogPrimitives()   (G_GnlEnv->UseVerilogPrimitives)
#define GnlEnvPostOpt()                (G_GnlEnv->DoPostOpt)
#define GnlEnvVDD()                   (G_GnlEnv->VDD)
#define GnlEnvPrintResizedGates()       (G_GnlEnv->PrintResizedGates)
#define GnlEnvReplaceOnlyCombiCompo()  (G_GnlEnv->ReplaceOnlyCombiCompo)
#define GnlEnvRemoveBuffer()            (G_GnlEnv->RemoveBuffer)
#define GnlEnvDontTouch()              (G_GnlEnv->DontTouch)

/* Options For VERIFICATION */
#define GnlEnvReference()              (G_GnlEnv->Reference)
#define GnlEnvMaxBddNode()              (G_GnlEnv->MaxBddNode)
#define GnlEnvInputAssoc()              (G_GnlEnv->InputAssoc)
#define GnlEnvOutputAssoc()             (G_GnlEnv->OutputAssoc)
#define GnlEnvXorUnproved()             (G_GnlEnv->XorUnproved)
#define GnlEnvIgnoreRandomSim()         (G_GnlEnv->IgnoreRandomSim)

#define SetGnlEnvMode(x)               (G_GnlEnv->Mode = x)
#define SetGnlEnvInput(x)              (G_GnlEnv->Input = x)
#define SetGnlEnvInputFormat(x)         (G_GnlEnv->InputFormat = x)
#define SetGnlEnvOutput(x)              (G_GnlEnv->Output = x)
#define SetGnlEnvLog(x)                (G_GnlEnv->Log = x)

```

gnloption.h

```
#define SetGnlEnvOptForce(x)          (G_GnlEnv->OptForce = x)
#define SetGnlEnvCriterion(x)         (G_GnlEnv->Criterion = x)
#define SetGnlEnvUseQBar(x)           (G_GnlEnv->UseQBar = x)
#define SetGnlEnvPinSeqPolMatching(x) (G_GnlEnv->PinSeqPolMatching = x)
#define SetGnlEnvInject(x)            (G_GnlEnv->Inject = x)
#define SetGnlEnvFlatten(x)           (G_GnlEnv->Flatten = x)
#define SetGnlEnvFlattenStr(x)        (G_GnlEnv->FlattenStr = x)
#define SetGnlEnvFlatListModules(x)   (G_GnlEnv->FlatListModules = x)
#define SetGnlEnvFlattenInstanceName(x) (G_GnlEnv->FlattenInstanceName = x)
#define SetGnlEnvLib(x)               (G_GnlEnv->Lib = x)
#define SetGnlEnvTop(x)               (G_GnlEnv->Top = x)
#define SetGnlEnvModelDir(x)          (G_GnlEnv->ModelDir = x)
#define SetGnlEnvModelExt(x)          (G_GnlEnv->ModelExt = x)
#define SetGnlEnvBuildLibForce(x)     (G_GnlEnv->BuildLibForce = x)
#define SetGnlEnvFsmDir(x)            (G_GnlEnv->FsmDir = x)
#define SetGnlEnvPrintHierarchy(x)    (G_GnlEnv->PrintHierarchy = x)
#define SetGnlEnvReportData(x)        (G_GnlEnv->ReportData = x)
#define SetGnlEnvPrintNbCritPath(x)   (G_GnlEnv->PrintNbCritPath = x)
#define SetGnlEnvFoldIsoPart(x)       (G_GnlEnv->FoldIsoPart = x)
#define SetGnlEnvPrintMaxFanoutVars(x) (G_GnlEnv->PrintMaxFanoutVars = x)
#define SetGnlEnvRespectLibConstraints(x) \
    (G_GnlEnv->RespectLibConstraints = x)
#define SetGnlEnvPrintClkDomain(x)    \
    (G_GnlEnv->PrintClkDomain = x)
#define SetGnlEnvPrintCritRegion(x)   \
    (G_GnlEnv->PrintCritRegion = x)
#define SetGnlEnvMaxCellSizeSupport(x) \
    (G_GnlEnv->MaxCellSizeSupport = x)
#define SetGnlEnvMinShareLogicSize(x) \
    (G_GnlEnv->MinShareLogicSize = x)
#define SetGnlEnvUseVerilogPrimitives(x) \
    (G_GnlEnv->UseVerilogPrimitives = x)
#define SetGnlEnvPostOpt(x)           (G_GnlEnv->DoPostOpt = x)
#define SetGnlEnvVDD(x)               (G_GnlEnv->VDD = x)
#define SetGnlEnvPrintResizedGates(x) (G_GnlEnv->PrintResizedGates = x)
#define SetGnlEnvReplaceOnlyCombiCompo(x) \
    (G_GnlEnv->ReplaceOnlyCombiCompo = x)
#define SetGnlEnvRemoveBuffer(x)      (G_GnlEnv->RemoveBuffer = x)
#define SetGnlEnvDontTouch(x)         (G_GnlEnv->DontTouch = x)

/* Options For VERIFICATION */
#define SetGnlEnvReference(x)         (G_GnlEnv->Reference = x)
#define SetGnlEnvMaxBddNode(x)         (G_GnlEnv->MaxBddNode = x)
#define SetGnlEnvInputAssoc(x)         (G_GnlEnv->InputAssoc = x)
#define SetGnlEnvOutputAssoc(x)        (G_GnlEnv->OutputAssoc = x)
#define SetGnlEnvXorUnproved(x)        (G_GnlEnv->XorUnproved = x)
#define SetGnlEnvIgnoreRandomSim(x)   (G_GnlEnv->IgnoreRandomSim = x)

/* ----- */
/* GNL_OPTION */
/* ----- */
typedef struct GNL_OPTION_STRUCT
{
    char *Name;
    char *Alias;
    char *OptionValue;
    char *Help1;
```

gnloption.h

```
    char          *Help2;
    char          *Help3;
    char          *Help4;
    char          *Help5;
    void          (*Function) ();
    int           OptionType;
}

GNL_OPTION_REC, *GNL_OPTION;

/* ----- EOF ----- */
```

gnlparse.h

```
/*-----*/
/*
 *      File:          gnlparse.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#ifndef GNLPARSE_H
#define GNLPARSE_H

#include <stdio.h>

#include "bbdd.h"           /* for type BDD */           */
#include "blist.h"          /* for type BLIST */          */
/*-----*/
#define MAX_IN            100000
#define MAX_OUT           100000
#define MAX_LOC            100000
#define MAX_ARG             100

#define MAX_EQU        (MAX_OUT+MAX_LOC)
#define MAX_VARS        (MAX_IN+MAX_OUT+MAX_LOC)

/*-----*/
/* IO_TYPE */
/*-----*/
typedef enum {
    NL_IN,
    NL_OUT,
    NL_LOC
} NL_IO;

/*-----*/
/* NODE_OP */
/*-----*/
typedef enum {
    /* RTL HDL usefull operators */
    BIT_OP,           /* 1-ary operator: its son is a 1-bit signal */
    BUS_OP,           /* n-ary op: (name, idxLeft, IdxRight) */
    IF_OP,
    CASE_OP,
    FOR_OP,
    WHILE_OP,
    SEQ_OP,
    INST_OP,
    TICK_OP,
    EQU_OP,           /* = */
    GT_OP,            /* > */
    LT_OP,            /* < */
    GTE_OP,           /* >= */
    LTE_OP,           /* <= */
    ASSIGN_OP,
    /* NL netlist usefull operators */
}
```

gnlparse.h

```

IDENT_OP,    /* Identity operator: ex primary output is a      */
/* direct primary input: Out = IDENT (In) */          */
/* 1 ary-op.                                         */
AND_OP,      /* n-ary associative operator                      */
OR_OP,       /* n-ary associative operator                      */
NAND_OP,     /* n-ary associative operator                      */
NOR_OP,      /* n-ary associative operator                      */
NOT_OP,      /* n-ary associative operator                      */
XOR_OP,      /* n-ary associative operator                      */
XNOR_OP,     /* n-ary associative operator                      */
MUX_OP,      /* n-ary associative operator                      */

BUF_OP,      /* 1-ary op: q = d                                */
TRISTATE_OP, /* 2-ary op: q = f (d, en)                      */
DFF_OP,      /* 6-ary op: q = f (d, ck, edg, rst, set, en) */
LATCH_OP,    /* 6-ary op: q = f (d, ck, lvl, rst, set, en) */
JOINT_OP,    /* n-ary op: joint of several incoming nets      */

} NODE_OP;

/*-----*/
/* NL_VALUE */
/*-----*/
typedef enum {
    NL_U, /* Unknown */                                     */
    NL_X, /* Conflict */                                    */
    NL_0, /* Logical 0 */                                   */
    NL_1, /* Logical 1 */                                   */
    NL_Z /* High impedance */                           */
} NL_VALUE;

/*-----*/
/* NL_JUSTIFY */
/*-----*/
typedef enum {
    JUSTIFY_INPUT,
    JUSTIFY_OR,
    JUSTIFY_AND,
    JUSTIFY_XOR
} NL_JUSTIFY_OP;

typedef struct NL_JUSTIFY_STRUCT {
    int             Var; /* Of type NL_VAR */
    NL_JUSTIFY_OP  Op;
    BLIST          Sons;
    BLIST          Dads;
    int            Off;
} NL_JUSTIFY_REC, *NL_JUSTIFY;

/*-----*/
/* NL_VAR type */
/*-----*/
/* This structure represents a primary in/out or local variable "Var" */
/*-----*/
typedef struct NL_VAR_STRUCT {
    char           *Name; /* Name of the Var */
}

```

gnlpars.h

```
int          Id;      /* Unique ID identifying the Var      */
NL_IO       Dir;      /* NL_IN, NL_OUT or NL_LOC           */
int          NbDads;  /* Number of NL_NODE using this Var   */
int          *Dads;    /* Array of NL_NODE using this Var   */
int          Node;    /* If NL_IN or NL_LOC it is its function*/
                  /* Externally this field is viewed as */
                  /* type NL_NODE.                      */
BDD         Bdd;     /* Bdd function associated to this VAR */
int          BddIndex;
BDD         VarBdd;
NL_VALUE    Value;   /* NL_VALUE associated to this Var   */
int          NbVarNodes; /* Nb bdd nodes with this var       */
int          Hight;   /* Hight of the Var in the Netlist   */
int          Depth;   /* Depth of the Var in the netlist   */
int          Tag;     /* Tag used for traversal algorithms */
int          TagBdd;
int          Nl;      /* Pointer on the Nl netlist it belongs */
int          Hook;    /* Associated Var in the other netlist */
int          NbBddNodes;
int          SupportSize;
int          AndSet;
int          Line;    /* Line number in the original file */
} NL_VAR_REC;

typedef NL_VAR_REC *NL_VAR;

/*-----
/* NL_NODE type
/*-----*/
/* This structure represents the node of a nl netlist.
/*-----*/
typedef struct NL_NODE_STRUCT {
    NL_VAR          Var;
    NODE_OP         Op;
    int             NbSons;
    NL_VAR          *Sons;
    BDD            Bdd;
    int             Tag;
    int             TagBdd;
    NL_VALUE        Value;
    int             Hight;
    int             Hook;
} NL_NODE_REC;

typedef NL_NODE_REC *NL_NODE;

#define GetNlNodeVar(nln)          ((nln)->Var)
#define GetNlNodeOp(nln)           ((nln)->Op)
#define GetNlNodeNbSons(nln)       ((nln)->NbSons)
#define GetNlNodeSons(nln)         ((nln)->Sons)
#define GetNlNodeBdd(nln)          ((nln)->Bdd)
#define GetNlNodeTag(nln)          ((nln)->Tag)
#define GetNlNodeTagBdd(nln)       ((nln)->TagBdd)
#define GetNlNodeValue(nln)        ((nln)->Value)
#define GetNlNodeHight(nln)        ((nln)->Hight)
#define GetNlNodeHook(nln)         ((nln)->Hook)
```

gnlpars.h

```

#define SetNlNodeVar(nln, V)          ((nln)->Var = V)
#define SetNlNodeOp(nln, O)           ((nln)->Op = O)
#define SetNlNodeNbSons(nln, N)       ((nln)->NbSons = N)
#define SetNlNodeSons(nln, S)         ((nln)->Sons = S)
#define SetNlNodeBdd(nln, B)          ((nln)->Bdd = B)
#define SetNlNodeTag(nln, T)          ((nln)->Tag = T)
#define SetNlNodeTagBdd(nln, T)       ((nln)->TagBdd = T)
#define SetNlNodeValue(nln, B)         ((nln)->Value = B)
#define SetNlNodeHight(nln, B)         ((nln)->Hight = B)
#define SetNlNodeHook(nln, H)          ((nln)->Hook = H)

/*-----*/
/* get and Set of type NL_VAR */
/*-----*/
#define GetNlVarName(nlv)            ((nlv)->Name)
#define GetNlVarId(nlv)             ((nlv)->Id)
#define GetNlVarDir(nlv)            ((nlv)->Dir)
#define GetNlVarNode(nlv)            ((NL_NODE)((nlv)->Node))
#define GetNlVarBdd(nlv)             ((nlv)->Bdd)
#define GetNlVarValue(nlv)           ((nlv)->Value)
#define GetNlVarImply(nlv)           ((nlv)->Imply)
#define GetNlVarHight(nlv)           ((nlv)->Hight)
#define GetNlVarDepth(nlv)           ((nlv)->Depth)
#define GetNlVarTag(nlv)             ((nlv)->Tag)
#define GetNlVarTagBdd(nlv)          ((nlv)->TagBdd)
#define GetNlVarNl(nlv)              ((nlv)->Nl)
#define GetNlVarHook(nlv)             ((nlv)->Hook)
#define GetNlVarLine(nlv)            ((nlv)->Line)

#define SetNlVarName(nlv,N)          ((nlv)->Name = N)
#define SetNlVarId(nlv,I)            ((nlv)->Id = I)
#define SetNlVarDir(nlv,D)           ((nlv)->Dir = D)
#define SetNlVarNode(nlv,N)          ((nlv)->Node = (int)N)
#define SetNlVarBdd(nlv,B)           ((nlv)->Bdd = B)
#define SetNlVarValue(nlv,V)          ((nlv)->Value = V)
#define SetNlVarImply(nlv,V)         ((nlv)->Imply = V)
#define SetNlVarHight(nlv,V)          ((nlv)->Hight = V)
#define SetNlVarDepth(nlv,V)          ((nlv)->Depth = V)
#define SetNlVarTag(nlv,T)            ((nlv)->Tag = T)
#define SetNlVarTagBdd(nlv,T)         ((nlv)->TagBdd = T)
#define SetNlVarNl(nlv,N)            ((nlv)->Nl = N)
#define SetNlVarHook(nlv,B)           ((nlv)->Hook = B)
#define SetNlVarLine(nlv,L)           ((nlv)->Line = L)

/*-----*/
/* NL_NODE_CELL */
/*-----*/
typedef struct NL_NODE_CELL_STRUCT {
    NL_NODE                           Node;
    struct NL_NODE_CELL_STRUCT      *Next;
} NL_NODE_CELL_REC;

typedef NL_NODE_CELL_REC *NL_NODE_CELL;

/*-----*/

```

gnlpars.h

```

/* NL_VAR_CELL type */  

/*-----*/  

typedef struct NL_VAR_CELL_STRUCT {  

    NL_VAR Var;  

    struct NL_VAR_CELL_STRUCT *Next;  

} NL_VAR_CELL_REC;  

typedef NL_VAR_CELL_REC *NL_VAR_CELL;  

#define GetNlVarCellVar(nlv) ((nlv)->Var)  

#define GetNlVarCellNext(nlv) ((nlv)->Next)  

#define SetNlVarCellVar(nlv,N) ((nlv)->Var = N)  

#define SetNlVarCellNext(nlv,N) ((nlv)->Next = N)  

/*-----*/  

/* NL_CONSTRAINT */  

/*-----*/  

typedef struct NL_CONSTRAINT_STRUCT {  

    NL_VAR Var;  

    NL_VALUE Value;  

} NL_CONSTRAINT_REC;  

typedef NL_CONSTRAINT_REC *NL_CONSTRAINT;  

/*-----*/  

/* NL type */  

/*-----*/  

typedef struct NL_STRUCT {  

    int Tag; /* Internal purpose: for traversal */  

    int TagBdd; /* To inform if the current Bdd field of*/  

                 /* a Var or Node is valid. */  

    char *Name;  

    int UniqueId;  

    int NbInputs;  

    int NbOutputs;  

    int NbLocals;  

    NL_VAR *VarIns; /* Of size NbInputs */  

    NL_VAR *VarOuts; /* Of size NbOutputs */  

    NL_VAR *VarLocs; /* Of Size VarLocs */  

    NL_VAR *Vars; /* Var[I] gives the Var which Id is I */  

} NL_REC;  

typedef NL_REC *NL;  

#define GetNlName(nl) ((nl)->Name)  

#define GetNlUniqueId(nl) ((nl)->UniqueId)  

#define GetNlNbInputs(nl) ((nl)->NbInputs)  

#define GetNlNbOutputs(nl) ((nl)->NbOutputs)  

#define GetNlNbLocals(nl) ((nl)->NbLocals)  

#define GetNlVarIN(nl) ((nl)->VarIns)  

#define GetNlVarOUT(nl) ((nl)->VarOuts)  

#define GetNlVarLOC(nl) ((nl)->VarLocs)  

#define GetNlVarWithId(nl) ((nl)->Vars)

```

```

#define SetNlName(nl, N)          ((nl)->Name = N)
#define SetNlUniqueId(nl, I)       ((nl)->UniqueId = I)
#define SetNlNbInputs(nl, n)       ((nl)->NbInputs = n)
#define SetNlNbOutputs(nl, n)      ((nl)->NbOutputs = n)
#define SetNlNbLocals(nl, n)       ((nl)->NbLocals = n)
#define SetNlVarIN(nl, In)         ((nl)->VarIns = In)
#define SetNlVarOUT(nl, Out)       ((nl)->VarOuts = Out)
#define SetNlVarLOC(nl, Loc)       ((nl)->VarLocs = Loc)
#define SetNlVarWithId(nl, Id)     ((nl)->Vars = Id)

/*-----*/
/* Type NL_STATUS */
/*-----*/
/* ERROR returned codes during NL construction and manipulation. */
/*-----*/
typedef enum {

    NL_OK,                      /* Everything OK ! */
    NL_MEMORY_FULL               /* No more memory available */

} NL_STATUS;

/*----- EOF -----*/
#endif

```

gnlparsel

```
%{
/*-----*/
/*
 *      File:          gnlparse.l
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: non
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>
#include <strings.h>
#include "tokens.h"

#define COMPO_ARE_ALL_USER

char      namebuf[256];      /* to store temporaly the string      */
extern int  gnllineno;
int       IfDefMode; /* If Def mode activated then everything*/
                     /* is considered as comment           */

static int  IsUserCompo = 0; /* if 'IsUserCompo' is 1 then all prede-*/
                           /* fined verilog gates like 'and', 'or'   */
                           /* are considered and processed as user   */
                           /* components.                         */
}

VALUE      [0-9][0-9]*[''][a-zA-Z][0-9]*
NAME       [a-zA-Z_][0-9a-zA-Z_\-\.\@\_]*[\\][\\{}\\\"$](~><+=,;\\-:@#*\|\!\?_\[\]\0-9a-zA-Z_]+
ANYNAME    [\\][\\{}\\\"$](~><+=,;\\-:@#*\|\!\?_\0-9a-zA-Z_\t)*
COMMENT    [//][\\{}\\\"$](~><+=,;\\-:@#*\|\!\?_\0-9a-zA-Z_\t)*
ATTRIBUTE  [\`][\"$](~><+=,;\\-:@#*\|\!\?_\[\]\0-9a-zA-Z_\t)*
INTEGER    [0-9][0-9]*
CR         [\n]
BLANK     [ \t]

%%
=          {
    if (!IfDefMode)
        return ((int) '=');
}

\.
{
    if (!IfDefMode)
        return ((int) '.');
}

\(
{
    if (!IfDefMode)
```

gnlparsel

```
        return ((int) '(');
    }

\\{
    if (!IfDefMode)
        return ((int) ')');

\\[
    if (!IfDefMode)
        return ((int) '[');

\\]
    if (!IfDefMode)
        return ((int) ']');

\\,
    if (!IfDefMode)
        return ((int) ',');

\\;
    if (!IfDefMode)
        return ((int) ';');

\\:
    if (!IfDefMode)
        return ((int) ':');

\\#
    if (!IfDefMode)
        return ((int) '#');

\\{
    if (!IfDefMode)
        return ((int) '{');

\\}
    if (!IfDefMode)
        return ((int) '}');

{ INTEGER}
{
    if (!IfDefMode)
    {
        gnllval.ival = atoi (gnltext);
        return (G_INTEGER);
    }
}

input {
```

gnlparsel

```
        if (!IfDefMode)
            return (G_INPUT);
    }

supply1
{
    if (!IfDefMode)
        return (G_WIRE);
}

supply0
{
    if (!IfDefMode)
        return (G_WIRE);
}

output
{
    if (!IfDefMode)
        return (G_OUTPUT);
}

inout
{
    if (!IfDefMode)
        return (G_INOUT);
}

(and)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {
            strcpy (namebuf,gnltext);
            gnllval.sval = namebuf;
            return (G_NAME);
        }
        else
        {
            gnllval.ival = G_AND;
            return (G_AND);
        }
    }
}

(nand)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {
            strcpy (namebuf,gnltext);
            gnllval.sval = namebuf;
            return (G_NAME);
        }
        else
        {
            gnllval.ival = G_NAND;
            return (G_NAND);
        }
    }
}
```

```

(or)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {
            strcpy (namebuf,gnltext);
            gnllval.sval = namebuf;
            return (G_NAME);
        }
        else
        {
            gnllval.ival = G_OR;
            return (G_OR);
        }
    }
}

(nor)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {
            strcpy (namebuf,gnltext);
            gnllval.sval = namebuf;
            return (G_NAME);
        }
        else
        {
            gnllval.ival = G_NOR;
            return (G_NOR);
        }
    }
}

(xor)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {
            strcpy (namebuf,gnltext);
            gnllval.sval = namebuf;
            return (G_NAME);
        }
        else
        {
            gnllval.ival = G_XOR;
            return (G_XOR);
        }
    }
}

(xnor)
{
    if (!IfDefMode)
    {
        if (IsUserCompo)
        {

```

gnlparsel

```
        strcpy (namebuf,gnltext);
        gnllval.sval = namebuf;
        return (G_NAME);
    }
    else
    {
        gnllval.ival = G_XNOR;
        return (G_XNOR);
    }
}
}

(not)
{
if (!IfDefMode)
{
    if (IsUserCompo)
    {
        strcpy (namebuf,gnltext);
        gnllval.sval = namebuf;
        return (G_NAME);
    }
    else
    {
        gnllval.ival = G_NOT;
        return (G_NOT);
    }
}
}

dffx
{
if (!IfDefMode)
{
    gnllval.ival = G_DFFX;
    return (G_DFFX);
}
}

dff1
{
if (!IfDefMode)
{
    gnllval.ival = G_DFF1;
    return (G_DFF1);
}
}

dff0
{
if (!IfDefMode)
{
    gnllval.ival = G_DFF0;
    return (G_DFF0);
}
}

dff
{
if (!IfDefMode)
{
```

gnlparsel

```
        gnllval.ival = G_DFF;
        return (G_DFF);
    }
}

latch
{
    if (!IfDefMode)
    {
        gnllval.ival = G_LATCH;
        return (G_LATCH);
    }
}

latchx
{
    if (!IfDefMode)
    {
        gnllval.ival = G_LATCHX;
        return (G_LATCHX);
    }
}

latch1
{
    if (!IfDefMode)
    {
        gnllval.ival = G_LATCH1;
        return (G_LATCH1);
    }
}

latch0
{
    if (!IfDefMode)
    {
        gnllval.ival = G_LATCH0;
        return (G_LATCH0);
    }
}

tristate
{
    if (!IfDefMode)
    {
        gnllval.ival = G_TRISTATE;
        return (G_TRISTATE);
    }
}

buf
{
    if (!IfDefMode)
    {
        gnllval.ival = G_BUF;
        return (G_BUF);
    }
}

module
{
    if (!IfDefMode)
        return (G_MODULE);
```

gnlparsel.l

```
        }

endmodule
{
    if (!IfDefMode)
        return (G_ENDMODULE);
}

assign
{
    if (!IfDefMode)
        return (G_ASSIGN);
}

wire
{
    if (!IfDefMode)
        return (G_WIRE);
}

tri
{
    if (!IfDefMode)
        return (G_WIRE);
}

reg
{
    if (!IfDefMode)
        return (G_WIRE);
}

1'b0
{
    if (!IfDefMode)
    {
        strcpy (namebuf,"0");
        gnllval.sval = namebuf;
        return (G_BIT_0);
    }
}

1'b1
{
    if (!IfDefMode)
    {
        strcpy (namebuf,"1");
        gnllval.sval = namebuf;
        return (G_BIT_1);
    }
}

{CR}
{
    gnlllineno++;
}

{COMMENT}
{}

{ATTRIBUTE}
{
    if (GnlIsIfDef (gnltext))
    {
        IfDefMode = 1;
        return (G_IFDEF);
    }
    if (GnlIsEndIf (gnltext))
    {
        IfDefMode = 0;
        return (G_ENDIF);
    }
}
```

gnlparsel

```
        }
    {NAME}
    {
        if (!IfDefMode)
        {
            strcpy (namebuf,gnltext);
            gnlval.sval = namebuf;
            return (G_NAME);
        }
    }
{ANYNAME}
{
    if (!IfDefMode)
    {
        strcpy (namebuf,gnltext);
        gnlval.sval = namebuf;
        return (G_NAME);
    }
}

{VALUE}
{
    if (!IfDefMode)
    {
        strcpy (namebuf,gnltext);
        gnlval.sval = namebuf;
        return (G_NAME);
    }
}
.
;

%%

/*-----
/* yywrap
/*-----
yywrap ()
{
    return (1);
}

/*-----
/* GnlIsIfDef
/*-----
int GnlIsIfDef (String)
    char      *String;
{
    if (strlen (String) < strlen ("`ifdef"))
        return (0);

    if ((String[0] == '`') &&
        (String[1] == 'i') &&
        (String[2] == 'f') &&
        (String[3] == 'd') &&
        (String[4] == 'e') &&
        (String[5] == 'f'))
        return (1);

    return (0);
}
```

gnlparse.l

```
}
```

```
/*-----*/  
/* GnlIsEndIf */  
/*-----*/  
int GnlIsEndIf (String)  
    char      *String;  
{  
    if (strlen (String) < strlen ("`endif"))  
        return (0);  
  
    if ((String[0] == '`') &&  
        (String[1] == 'e') &&  
        (String[2] == 'n') &&  
        (String[3] == 'd') &&  
        (String[4] == 'i') &&  
        (String[5] == 'f'))  
        return (1);  
  
    return (0);  
}  
  
/*-----*/  
/*-----*/
```

gnlpars.y

```
%{
/*-----*/
/*
 *      File:          gnlparse.y
 *      Version:        1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>
#include <strings.h>

#include "blist.h"
#include "gnl.h"

/*-----*/
/* ERROR/WARNING MESSAGES */
/*-----*/
#define ERR_NEVER_DECLARED \
    " Error (l.%d): Signal <%s> is not declared\n"
#define ERR_PORT_REDECLARED \
    " ERROR (l.%d): Port <%s> already declared\n"
#define WARN_NEVER_DECLARED \
    " WARNING (l.%d): Signal <%s> is not declared\n"
#define WARN_DO_NOT_ANALYZE \
    " WARNING (l.%d): `ifdef part not analyzed\n"

/*-----*/
/* EXTERN */
/*-----*/
int          gnllinenr;
extern char*   gnltext;
GNL_STATUS   GnlStatus;
extern BLIST    G_ListOfGnls;      /* The main list storing all the GNLS
 */
extern GNL   G_CurrentGnl;       /* The current Gnl/Module we process      */
extern int   G_GnlIsBig;
extern int   IfDefMode;

/*-----*/
/* GLOBAL VARIABLES */
/*-----*/
static GNL_VAR          Var;
static GNL_VAR          VarRight;
static int              VarMsb;           /* Msb of current Var.                  */
static int              VarLsb;           /* Lsb of current Var.                  */
static char*            *NewName;
static GNL_FUNCTION      NewFunction;
static BLIST             VarListAssoc;
static BLIST             ListDefParam;
static char*            *InstanceName;
```

gnlpase.y

```
static GNL_NODE          CstNode;
static GNL_NODE          NewNode;
static char              *UserGateName;
static char              StoreUserName[528];
static GNL_ASSOC          NewAssoc;
static BLIST              ListConcat;
static char              *FormalName;
static int               SignalType;
static int               GateOp;
static BLIST              ListPorts;

#define INPUT_PORT          0
#define OUTPUT_PORT         1
#define LOCAL_SIGNAL        2
#define INOUT_PORT          3

/*-----*/
%
```

```
%union      {
    int    ival;
    char   *sval;
}

%start verilog_nl

/* Types for non-terminals. */
%type <sval>      port
%type <sval>      signal
%type <ival>       formal_signal
%type <sval>       instance_id

%type <ival>       signal_type
%type <ival>       msb
%type <ival>       lsb
%type <ival>       gate
%type <ival>       def_param
%type <ival>       user_gate

/* Types for terminals */
%token   <ival>     G_INPUT
%token   <ival>     G_OUTPUT

%token   <ival>     G_AND
%token   <ival>     G_OR
%token   <ival>     G_NAND
%token   <ival>     G_NOR
%token   <ival>     G_NOT
%token   <ival>     G_PADD
%token   <ival>     G_PCOMP
%token   <ival>     G_XOR
%token   <ival>     G_XNOR
%token   <ival>     G_MUX
%token   <ival>     G_BUFF
%token   <ival>     G_TRISTATE
```

gnlpars.y

```
%token      <ival>      G_DFF
%token      <ival>      G_DFFX
%token      <ival>      G_DFF1
%token      <ival>      G_DFF0
%token      <ival>      G_LATCH
%token      <ival>      G_LATCHX
%token      <ival>      G_LATCH1
%token      <ival>      G_LATCH0
%token      <ival>      G_MODULE
%token      <ival>      G_ENDMODULE
%token      <ival>      G_ASSIGN
%token      <ival>      G_INTEGER
%token      <ival>      G_WIRE
%token      <ival>      G_REG

%token      <sval>      G_BIT_0
%token      <sval>      G_BIT_1
%token      <sval>      G_NAME

%token      <ival>      G_INOUT
%token      <ival>      G_USER_GATE
%token      <ival>      G_IFDEF
%token      <ival>      G_ENDIF
%token      <ival>      G_BUF

%%

/*-----*/
/* GNL PARSER */
/*-----*/
/* The verilog file is a list of verilog modules.          */
/* The current built gnl is 'G_CurrentGnl' and is added in global list */
/* 'G_ListOfGnls'.                                         */
/*-----*/
verilog_nl :    {
                    gnllineno = 1; /* Initialize at 1rst line     */
                    IfDefMode = 0;
                    if (BListCreate (&G_ListOfGnls))
                        return (GNL_MEMORY_FULL);
                }
                list_module_def
;

list_module_def : /* no modules      */
| module_def
{
    if (BListAddElt (G_ListOfGnls, (int)G_CurrentGnl))
        return (GNL_MEMORY_FULL);
}
list_module_def
;

module_def : G_MODULE G_NAME
{
    fprintf (stderr, "    o Analyzing module [%s]\n",

```

```

        $2);
    if (G_GnlIsBig)
    {
        if (GnlCreate ($2, &G_CurrentGnl))
            return (GNL_MEMORY_FULL);
        }
    else
    {
        if (GnlSmallCreate ($2, &G_CurrentGnl))
            return (GNL_MEMORY_FULL);
        }
    SetGnlLineNumber (G_CurrentGnl, gnllineno);

/* Adding var corresponding to Boolean cste '0'*/
if ((GnlStatus = GnlVarCreateAndAddInHashTable
        (G_CurrentGnl, "0", &Var)))
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (G_CurrentGnl, GNL_CONSTANTE,
        &CstNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (CstNode, (BLIST)0);
if (GnlFunctionCreate (G_CurrentGnl, Var, CstNode,
        &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (Var, NewFunction);

/* Adding var corresponding to Boolean cste '1'*/
if ((GnlStatus = GnlVarCreateAndAddInHashTable
        (G_CurrentGnl, "1", &Var)))
    return (GNL_MEMORY_FULL);
if (GnlCreateNode (G_CurrentGnl, GNL_CONSTANTE,
        &CstNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (CstNode, (BLIST)1);
if (GnlFunctionCreate (G_CurrentGnl, Var, CstNode,
        &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (Var, NewFunction);

        if (BListCreate (&ListPorts))
            return (GNL_MEMORY_FULL);
        SetGnlListPorts (G_CurrentGnl, ListPorts);
    }
(' list_of_ports ') ' ';
list_signals_decls
{
    if ((GnlStatus = GnlCreateAllSignals (G_CurrentGnl)))
        return (GnlStatus);
}
module_body
G_ENDMODULE
;

list_of_ports
|
list_of_ports1
;

```

gnlpars.y

```
list_of_ports1 : port
{
    /* We create a Var corresponding to the port. */
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
                    (G_CurrentGnl, $1, &Var)))
    {
        if (GnlStatus == GNL_VAR_EXISTS)
        {
            fprintf (stderr, ERR_PORT_REDECLARED,
                     gnllineno, GnlVarName (Var));
            return (GNL_VAR_EXISTS);
        }

        return (GNL_MEMORY_FULL);
    }
    if (SetGnlVarLineNumber (Var, gnllineno))
        return (GNL_MEMORY_FULL);
    SetGnlVarType (Var, GNL_VAR_ORIGINAL);
    if (BListAddElt (ListPorts, (int)Var))
        return (GNL_MEMORY_FULL);
}

| list_of_ports1 ',' port
{
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
                    (G_CurrentGnl, $3, &Var)))
    {
        if (GnlStatus == GNL_VAR_EXISTS)
        {
            fprintf (stderr, ERR_PORT_REDECLARED,
                     gnllineno, GnlVarName (Var));
            return (GNL_VAR_EXISTS);
        }

        return (GNL_MEMORY_FULL);
    }
    if (SetGnlVarLineNumber (Var, gnllineno))
        return (GNL_MEMORY_FULL);
    SetGnlVarType (Var, GNL_VAR_ORIGINAL);
    if (BListAddElt (ListPorts, (int)Var))
        return (GNL_MEMORY_FULL);
}
;

port : G_NAME
{
    $$ = $1;
}
;

list_signals_decls : /* empty list */
| list_signals_decls signal_decl
;
```

gnlpars.y

```
signal_decl : signal_type range list_signals_same_type ';' |  
    signal_type range G_NAME '=' G_NAME ';' {  
        if ((GnlStatus = GnlVarCreateAndAddInHashTable  
            (G_CurrentGnl, $3, &Var)))  
        {  
            if ((GnlStatus == GNL_VAR_EXISTS) &&  
                ($1 != LOCAL_SIGNAL))  
            {  
                /* Var exists and has already a range */  
                if (!GnlVarRangeUndefined (Var))  
                    GnlError (1);  
            }  
            /* No range so we set it. */  
            SetGnlVarMsb (Var, VarMsb);  
            SetGnlVarLsb (Var, VarLsb);  
            if ($1 == INPUT_PORT)  
                SetGnlVarDir (Var, GNL_VAR_INPUT);  
            else if ($1 == OUTPUT_PORT)  
                SetGnlVarDir (Var, GNL_VAR_OUTPUT);  
            else if ($1 == INOUT_PORT)  
                SetGnlVarDir (Var, GNL_VAR_INOUT);  
            if (SetGnlVarLineNumber (Var, gnllineno))  
                return (GNL_MEMORY_FULL);  
        }  
    ;  
  
list_signals_same_type : G_NAME {  
    if ((GnlStatus = GnlVarCreateAndAddInHashTable  
        (G_CurrentGnl, $1, &Var)))  
    {  
        if ((GnlStatus == GNL_VAR_EXISTS) &&  
            (SignalType != LOCAL_SIGNAL))  
        {  
            /* Var exists and has already a range*/  
            if (!GnlVarRangeUndefined (Var))  
                GnlError (1);  
        }  
        /* No range so we set it. */  
        SetGnlVarMsb (Var, VarMsb);  
        SetGnlVarLsb (Var, VarLsb);  
        if (SignalType == INPUT_PORT)  
            SetGnlVarDir (Var, GNL_VAR_INPUT);  
        else if (SignalType == OUTPUT_PORT)  
            SetGnlVarDir (Var, GNL_VAR_OUTPUT);  
        else if (SignalType == INOUT_PORT)  
            SetGnlVarDir (Var, GNL_VAR_INOUT);  
        /* Otherwise it is GNL_VAR_LOCAL */  
        if (SetGnlVarLineNumber (Var, gnllineno))  
            return (GNL_MEMORY_FULL);  
    }  
    | list_signals_same_type ',' G_NAME
```

```

{
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
        (G_CurrentGnl, $3, &Var)))
    {
        if ((GnlStatus == GNL_VAR_EXISTS) &&
            (SignalType != LOCAL_SIGNAL))
        {
            /* Var exists and has already a range*/
            if (!GnlVarRangeUndefined (Var))
                GnlError (1);
        }
    }
    /* No range so we set it. */
    SetGnlVarMsb (Var, VarMsb);
    SetGnlVarLsb (Var, VarLsb);
    if (SignalType == INPUT_PORT)
        SetGnlVarDir (Var, GNL_VAR_INPUT);
    else if (SignalType == OUTPUT_PORT)
        SetGnlVarDir (Var, GNL_VAR_OUTPUT);
    else if (SignalType == INOUT_PORT)
        SetGnlVarDir (Var, GNL_VAR_INOUT);
    /* Otherwise it is GNL_VAR_LOCAL */
}

if (SetGnlVarLineNumber (Var, gnllineno))
    return (GNL_MEMORY_FULL);
}
;

signal_type : G_INPUT   { SignalType = INPUT_PORT; }
| G_OUTPUT   { SignalType = OUTPUT_PORT; }
| G_INOUT    { SignalType = INOUT_PORT; }
| G_WIRE     { SignalType = LOCAL_SIGNAL; }
| G_REG      { SignalType = LOCAL_SIGNAL; }
;

module_body :      list_assigns_instances
;

list_assigns_instances :
| list_assigns_instances assign_or_instance
;

assign_or_instance :      assignment
| gate list_instance ';'
| G_IFDEF
{
    fprintf (stderr, WARN_DO_NOT_ANALYZE,
            gnllineno);
}
| G_ENDIF
;

assignment :      G_ASSIGN signal
{
    if ((GnlStatus =
        GnlGetVarFromName (G_CurrentGnl, $2, &Var)))

```

```

        {
            if ((GnlStatus == GNL_VAR_NOT_EXISTS))
            {
                if ((GnlStatus =
                    GnlVarCreateAndAddInHashTableWithNoTest
                    (G_CurrentGnl, $2, &Var)))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (GnlLocals (G_CurrentGnl),
                                Var))
                    return (GNL_MEMORY_FULL);
                SetGnlNbLocal (G_CurrentGnl,
                               GnlNbLocal (G_CurrentGnl)+1);
                if (SetGnlVarLineNumber (Var, gnllineno))
                    return (GNL_MEMORY_FULL);
            }
            else
                return (GNL_MEMORY_FULL);
        }
    }
    '=' signal
    {
        if ((GnlStatus =
            GnlGetVarFromName (G_CurrentGnl, $5, &VarRight)))
        {
            if ((GnlStatus == GNL_VAR_NOT_EXISTS))
            {
                if ((GnlStatus =
                    GnlVarCreateAndAddInHashTableWithNoTest
                    (G_CurrentGnl, $5, &VarRight)))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (GnlLocals (G_CurrentGnl),
                                VarRight))
                    return (GNL_MEMORY_FULL);
                SetGnlNbLocal (G_CurrentGnl,
                               GnlNbLocal (G_CurrentGnl)+1);
                if (SetGnlVarLineNumber (Var, gnllineno))
                    return (GNL_MEMORY_FULL);
            }
            else
                return (GNL_MEMORY_FULL);
        }
        if (GnlFunctionCreateFromVars (G_CurrentGnl, Var,
                                       VarRight,
                                       &NewFunction,
                                       gnllineno))
            return (GNL_MEMORY_FULL);
    }
    ;
;

list_instance      :      instance
|      list_instance ',' instance
;

instance      :      instance_id '('
{

```

```

        if (GnlStrCopy ($1, &InstanceName))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (2, &VarListAssoc))
            return (GNL_MEMORY_FULL);
    }
list_signals
')
{
/* In the case of G_USER_GATE we need to copy the */
/* name of the gate. */
if (GateOp == G_USER_GATE)
{
    if ((UserGateName =
        (char*)calloc (strlen (StoreUserGateName)+1,
                      sizeof(char))) == NULL)
        return (GNL_MEMORY_FULL);
    strcpy (UserGateName, StoreUserGateName);
}
if (GnlAnalyzeInstance (G_CurrentGnl, GateOp,
                        UserGateName,
                        InstanceName,
                        NULL, VarListAssoc,
                        gnllineno, 0))
    return (GNL_MEMORY_FULL);
}
|(
{
    if (BListCreateWithSize (2, &VarListAssoc))
        return (GNL_MEMORY_FULL);
}
list_signals
')
{
/* In the case of G_USER_GATE we need to copy the */
/* name of the gate. */
if (GateOp == G_USER_GATE)
{
    if ((UserGateName =
        (char*)calloc (strlen (StoreUserGateName)+1,
                      sizeof(char))) == NULL)
        return (GNL_MEMORY_FULL);
    strcpy (UserGateName, StoreUserGateName);
}
if (GnlAnalyzeInstance (G_CurrentGnl, GateOp,
                        UserGateName,
                        NULL,
                        NULL, VarListAssoc,
                        gnllineno, 0))
    return (GNL_MEMORY_FULL);
}
|
' #' '('
{
    if (BListCreateWithSize (1, &ListDefParam))
        return (GNL_MEMORY_FULL);
}
list_def_param
')' instance_id '('

```

```

{
    if (GnlStrCopy ($6, &InstanceName))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (2, &VarListAssoc))
        return (GNL_MEMORY_FULL);
}
list_signals
')
{
/* In the case of G_USER_GATE we need to copy the */
/* name of the gate. */
if (GateOp == G_USER_GATE)
{
    if ((UserGateName =
        (char*)calloc (strlen (StoreUserGateName)+1,
                      sizeof(char))) == NULL)
        return (GNL_MEMORY_FULL);
    strcpy (UserGateName, StoreUserGateName);
}
    if (GnlAnalyzeInstance (G_CurrentGnl, GateOp,
                           UserGateName,
                           InstanceName,
                           ListDefParam,
                           VarListAssoc, gnllineno, 0))
        return (GNL_MEMORY_FULL);
}

;

list_def_param : def_param
{
    if (BListAddElt (ListDefParam, $1))
        return (GNL_MEMORY_FULL);
}
| list_def_param ',' def_param
{
    if (BListAddElt (ListDefParam, $3))
        return (GNL_MEMORY_FULL);
}
;

def_param : G_INTEGER { $$ = $1; }

;

instance_id : G_NAME
;

gate : G_OR      {GateOp = G_OR;}
| G_AND      {GateOp = G_AND;}
| G_NOR      {GateOp = G_NOR;}
| G_NAND     {GateOp = G_NAND;}
| G_XOR      {GateOp = G_XOR;}
| G_XNOR     {GateOp = G_XNOR;}
| G_DFF      {GateOp = G_DFF;}
| G_DFFX     {GateOp = G_DFFX;}
| G_DFF1     {GateOp = G_DFF1;}
| G_DFF0     {GateOp = G_DFF0;}
;

```

```

    | G_MUX      {GateOp = G_MUX; }
    | G_LATCH    {GateOp = G_LATCH; }
    | G_LATCHX   {GateOp = G_LATCHX; }
    | G_LATCH1   {GateOp = G_LATCH1; }
    | G_LATCH0   {GateOp = G_LATCH0; }
    | G_TRISTATE {GateOp = G_TRISTATE; }
    | G_NOT      {GateOp = G_NOT; }
    | G_PADD     {GateOp = G_PADD; }
    | G_PCOMP    {GateOp = G_PCOMP; }
    | G_BUF      {GateOp = G_BUF; }
    | user_gate  {GateOp = G_USER_GATE; }

;

user_gate : G_NAME
{
    strcpy (StoreUserGateName, $1);
}
;

list_signals :
| list_signals1
;

list_signals1 : signal
{
    if (GnlGetVarFromName (G_CurrentGnl, $1, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
        {
            if ((GnlStatus =
                GnlVarCreateAndAddInHashTableWithNoTest
                (G_CurrentGnl, $1, &Var)))
                return (GNL_VAR_NOT_EXISTS);
            if (BListAddElt (GnlLocals (G_CurrentGnl), Var))
                return (GNL_MEMORY_FULL);
            SetGnlNbLocal (G_CurrentGnl,
                GnlNbLocal (G_CurrentGnl)+1);
            if (SetGnlVarLineNumber (Var, gnllineno))
                return (GNL_MEMORY_FULL);
        }
        else
            return (GNL_MEMORY_FULL);
    }
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, NULL);
    SetGnlAssocActualPort (NewAssoc, Var);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}
|
| '...' formal_signal
'(' '{'
{
    if (BListCreateWithSize (1, &ListConcat))
        return (GNL_MEMORY_FULL);
}
concat_signal
;
```

```

{
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    if (GnlCreateConcatNode (&NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (NewNode, ListConcat);
    SetGnlNodeLineNumber (NewNode, gnllineno);
    /* we define a concat node as actual port.          */
    SetGnlAssocActualPort (NewAssoc, (GNL_VAR)NewNode);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}
'.' '
| '.' formal_signal
(' signal ')
{
    if (GnlGetVarFromName (G_CurrentGnl, $4, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
        {
            if ((GnlStatus =
                GnlVarCreateAndAddInHashTableWithNoTest
                (G_CurrentGnl, $4, &Var)))
                return (GNL_VAR_NOT_EXISTS);
            if (BListAddElt (GnlLocals (G_CurrentGnl), Var))
                return (GNL_MEMORY_FULL);
            SetGnlNbLocal (G_CurrentGnl,
                            GnlNbLocal (G_CurrentGnl)+1);
            if (SetGnlVarLineNumber (Var, gnllineno))
                return (GNL_MEMORY_FULL);
        }
        else
            return (GNL_MEMORY_FULL);
    }
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    SetGnlAssocActualPort (NewAssoc, Var);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}
| '.' formal_signal
(' ')
{
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    SetGnlAssocActualPort (NewAssoc, NULL);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}
| list_signals1 ',' signal
{
    if (GnlGetVarFromName (G_CurrentGnl, $3, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)

```

```

{
    if ((GnlStatus =
        GnlVarCreateAndAddInHashTableWithNoTest
        (G_CurrentGnl, $3, &Var)))
        return (GNL_VAR_NOT_EXISTS);
    if (BListAddElt (GnlLocals (G_CurrentGnl), Var))
        return (GNL_MEMORY_FULL);
    SetGnlNbLocal (G_CurrentGnl,
                    GnlNbLocal (G_CurrentGnl)+1);
    if (SetGnlVarLineNumber (Var, gnllineno))
        return (GNL_MEMORY_FULL);
}
else
    return (GNL_MEMORY_FULL);
}
if (GnlCreateAssoc (&NewAssoc))
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort (NewAssoc, NULL);
SetGnlAssocActualPort (NewAssoc, Var);
if (BListAddElt (VarListAssoc, (int)NewAssoc))
    return (GNL_MEMORY_FULL);
}
| list_signals1 ',' '..' formal_signal
'(' '{'
{
    if (BListCreateWithSize (1, &ListConcat))
        return (GNL_MEMORY_FULL);
}
concat_signal
{
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    if (GnlCreateConcatNode (&NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlNodeSons (NewNode, ListConcat);
    SetGnlNodeLineNumber (NewNode, gnllineno);
    /* we define a concat node as actual port. */
    SetGnlAssocActualPort (NewAssoc, (GNL_VAR)NewNode);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
}
'} ')
| list_signals1 ',' '..' formal_signal
'(' signal ')'
{
    if (GnlGetVarFromName (G_CurrentGnl, $6, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
        {
            if ((GnlStatus =
                GnlVarCreateAndAddInHashTableWithNoTest
                (G_CurrentGnl, $6, &Var)))
                return (GNL_VAR_NOT_EXISTS);
            if (BListAddElt (GnlLocals (G_CurrentGnl), Var))
                return (GNL_MEMORY_FULL);
            SetGnlNbLocal (G_CurrentGnl,

```

```

                GnlNbLocal (G_CurrentGnl)+1);
        if (SetGnlVarLineNumber (Var, gnllineno))
            return (GNL_MEMORY_FULL);
        }
        else
            return (GNL_MEMORY_FULL);
    }
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    SetGnlAssocActualPort (NewAssoc, Var);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
    }
| list_signals1 ',' '.' formal_signal
'(' ')
{
    if (GnlCreateAssoc (&NewAssoc))
        return (GNL_MEMORY_FULL);
    SetGnlAssocFormalPort (NewAssoc, FormalName);
    SetGnlAssocActualPort (NewAssoc, NULL);
    if (BListAddElt (VarListAssoc, (int)NewAssoc))
        return (GNL_MEMORY_FULL);
    }
;
concat_signal : signal
{
    if (GnlGetVarFromName (G_CurrentGnl, $1, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr, ERR_NEVER DECLARED,
                     gnllineno, $1);
            GnlError (3);
            return (GNL_VAR_NOT_EXISTS);
        }
        return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (ListConcat, (int)Var))
        return (GNL_MEMORY_FULL);
}
| concat_signal ',' signal
{
    if (GnlGetVarFromName (G_CurrentGnl, $3, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr, ERR_NEVER DECLARED,
                     gnllineno, $3);
            GnlError (3);
            return (GNL_VAR_NOT_EXISTS);
        }
        return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (ListConcat, (int)Var))
        return (GNL_MEMORY_FULL);
}

```

gnlparse.y

```
        }

;

formal_signal      : G_NAME
{
    if (GnlStrCopy ($1, &FormalName))
        return (GNL_MEMORY_FULL);
}
| G_NAME '[' G_INTEGER ']'
{
    if ((GnlStatus = GnlStrIndexName ($1, $3, &FormalName)))
        return (GNL_MEMORY_FULL);
}
;

signal            : G_NAME
{
    $$ = $1;
}
| G_NAME '[' G_INTEGER ']'
{
    if ((GnlStatus = GnlStrIndexName ($1, $3, &NewName)))
        return (GnlStatus);
    $$ = NewName;
}
| G_BIT_0
{
    $$ = $1;
}
| G_BIT_1
{
    $$ = $1;
}
;

range             : /* no range      */
{
    VarMsb = VarLsb = -1;          /* No range a priori      */
}
| '[' msb ':' lsb ']'
{
    VarMsb = $2;
    VarLsb = $4;
}
;

msb               : G_INTEGER { $$ = $1; }
;
lsb               : G_INTEGER { $$ = $1; }
;
%%

/*-----*/
/* yyerror           */
/*-----*/
```

gnlpars.e.y

```
yyerror (s)
    char      *s;
{
    fprintf (stderr,"GNL-READER (l.%d): syntax error near '%s'\n",
            gnllinenr, gnltext);
    exit (1);
}
```

```
/*-----*/
```

gnlpart.c

```
/*-----*/
/*
 *      File:          gnlpart.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description: *
 */
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnloption.h"

#include "blist.e"

/*-----*/
/* GLOBAL VARIABLE */
/*-----*/
static int G_SupportId = 0;
extern GNL_ENV    G_GnlEnv;

/*-----*/
/* GnlSetSupportVarHook */
/*-----*/
void GnlSetSupportVarHook (Support)
    BLIST   Support;
{
    int             i;
    GNL_VAR  VarI;

    G_SupportId++;

    for (i=0; i<BListSize (Support); i++)
    {
        VarI = (GNL_VAR)BListElt (Support, i);
        SetGnlVarHook (VarI, G_SupportId);
    }
}

/*-----*/
/* GnlBListUnionSupport */
/*-----*/
GNL_STATUS GnlBListUnionSupport (SupportI, SupportJ, NbCommon, UnionList)
    BLIST   SupportI;
    BLIST   SupportJ;
```

gnlpart.c

```
int *NbCommon;
BLIST *UnionList;
{
    int i;
    GNL_VAR VarI;

    *NbCommon = 0;

    if (BListCreateWithSize (2, UnionList))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (SupportJ); i++)
    {
        VarI = (GNL_VAR)BListElt (SupportJ, i);
        if (GnlVarHook (VarI) == (void*)G_SupportId)
            (*NbCommon)++;
        else
        {
            if (BListAddElt ((*UnionList), (int)VarI))
                return (GNL_MEMORY_FULL);
        }
    }

    if (*NbCommon > 1)
    {
        for (i=0; i<BListSize (*UnionList); i++)
        {
            VarI = (GNL_VAR)BListElt (*UnionList, i);
            SetGnlVarHook (VarI, G_SupportId);
        }
        for (i=0; i<BListSize (SupportI); i++)
        {
            VarI = (GNL_VAR)BListElt (SupportI, i);
            if (BListAddElt ((*UnionList), (int)VarI))
                return (GNL_MEMORY_FULL);
        }
    }
    return (GNL_OK);
}

/****************************************
/* GnlMergeFunctions
*/
/****************************************
GNL_STATUS GnlMergeFunctions (ListSupports, ListFunctions,
                               MergedListOfSupports, ListOfListEquations)
{
    BLIST ListSupports;
    BLIST ListFunctions;
    BLIST *MergedListOfSupports;
    BLIST *ListOfListEquations;
    {
        int i;
        int j;
        int Added;
        GNL_FUNCTION FunctionI;
```

```

BLIST      SupportI;
GNL_FUNCTION FunctionJ;
BLIST      SupportJ;
BLIST      ListOfEquat;
BLIST      UnionList;
int        NbLit;
int        NbCommon;
int        Window;

if (BListCreateWithSize (1, MergedListOfSupports))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (1, ListOfListEquations))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListFunctions); i++)
{
    fprintf (stderr, "%c Merging [%d/%d]", 13, i,
             BListSize (ListFunctions));
    fflush (stderr);

FunctionI = (GNL_FUNCTION)BListElt (ListFunctions, i);
if (BListCopyNoEltCr ((BLIST)BListElt (ListSupports, i),
                      &SupportI))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (1, &ListOfEquat))
    return (GNL_MEMORY_FULL);
if (BListAddElt (ListOfEquat, (int)FunctionI))
    return (GNL_MEMORY_FULL);

/* We set the Var hook of each of the support 'SupportI' */
GnlSetSupportVarHook (SupportI);

/* Checks if functions after 'FunctionI' can be merged in the
/* same partition. */
Added = 1;
NbLit = GnlNodeNbLitt (GnlFunctionOnSet (FunctionI));
Window = i+20000;
while (Added)
{
    Added = 0;
    for (j=i+1; j<BListSize (ListFunctions); j++)
    {
        /* We stop if partition exceeds 8000 literals */
        if (NbLit > 8000)
            break;

        /* We stop if partition exceeds 400 Variables */
        if (BListSize (SupportI) > 400)
            break;

        if (j > Window)
            break;

        FunctionJ = (GNL_FUNCTION)BListElt (ListFunctions, j);
        SupportJ = (BLIST)BListElt (ListSupports, j);
    }
}

```

```

        if (GnlBListUnionSupport (SupportI, SupportJ,
                                  &NbCommon, &UnionList))
            return (GNL_MEMORY_FULL);
/*
    if (BListUnionNoEltCr (SupportI, SupportJ,
                           &UnionList))
        return (GNL_MEMORY_FULL);

NbCommon = BListSize (SupportI)+BListSize (SupportJ)-
           BListSize (UnionList);
*/
    if (NbCommon > 1)
    {
        if (BListAddElt (ListOfEquat, (int)FunctionJ))
            return (GNL_MEMORY_FULL);
        BListDelInsert (ListFunctions, j+1);
        BListDelInsert (ListSupports, j+1);
        BListQuickDelete (&SupportI);
        BListQuickDelete (&SupportJ);
        SupportI = UnionList;
        j--;
        Added = 1;
        NbLit += GnlNodeNbLitt (GnlFunctionOnSet (FunctionJ));
    }
    else
    {
        BListQuickDelete (&UnionList);
    }
}
}

if (BListAddElt (*MergedListOfSupports, (int)SupportI))
    return (GNL_MEMORY_FULL);
if (BListAddElt (*ListOfListEquations, (int)ListOfEquat))
    return (GNL_MEMORY_FULL);
}

if (BListSize (ListFunctions))
    fprintf (stderr, "%c Merging [%d/%d]\n", 13, i,
             BListSize (ListFunctions));

    return (GNL_OK);
}

/*-----*/
/* GnlGetFunctionSupportRec */ */
/*-----*/
GNL_STATUS GnlGetFunctionSupportRec (Node, Support)
    GNL_NODE      Node;
    BLIST        *Support;
{
    GNL_VAR  Var;
    BLIST    Sons;
    int       i;
    GNL_NODE SonI;

```

gnlpart.c

```
if (GnlNodeOp (Node) == GNL_CONSTANTE)
    return (GNL_OK);

if (GnlNodeOp (Node) == GNL_VARIABLE)
{
    Var = (GNL_VAR) GnlNodeSons (Node);
    if (!BListMemberOfList (Support, Var, IntIdentical))
        if (BListAddElt (Support, (int)Var))
            return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

Sons = GnlNodeSons (Node);
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE) BlistElt (GnlNodeSons (Node), i);
    if (GnlGetFunctionSupportRec (SonI, Support))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlGetFunctionSupport
/*-----*/
GNL_STATUS GnlGetFunctionSupport (Node, Support)
    GNL_NODE Node;
    BLIST *Support;
{

    if (BListCreateWithSize (1, Support))
        return (GNL_MEMORY_FULL);

    if (GnlGetFunctionSupportRec (Node, *Support))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlCreateGnlFromFunction
/*-----*/
GNL_STATUS GnlCreateGnlFromFunction (Gnl, ListEquations, Support, NewGnl)
    GNL Gnl;
    BLIST ListEquations;
    BLIST Support;
    GNL *NewGnl;
{
    int i;
    GNL_FUNCTION FunctionI;
    GNL_VAR VarI;
    BLIST NewList;
```

```

if ((*NewGnl = (GNL)GNL_CALLOC (1, sizeof(GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

/* The NewGnl inherits some common field from the original Gnl. One */
/* very important field to have in common is the same Hash Tabel of */
/* variables. */
SetGnlName (*NewGnl, GnlName (Gnl));
SetGnlHashNames (*NewGnl, GnlHashNames (Gnl));
SetGnlSourceFileName (*NewGnl, GnlSourceFileName (Gnl));
SetGnlListSourceFiles (*NewGnl, GnlListSourceFiles (Gnl));
SetGnlListPorts (*NewGnl, GnlListPorts (Gnl));

/* Creating List of nodes segments */
if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments ((*NewGnl), NewList);
SetGnlFirstNode ((*NewGnl), NULL);
SetGnlLastNode ((*NewGnl), NULL);

SetGnlInputs (*NewGnl, Support);
SetGnlNbIn (*NewGnl, BListSize (Support));

if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlOutputs ((*NewGnl), NewList);

for (i=0; i<BListSize (ListEquations); i++)
{
    FunctionI = (GNL_FUNCTION)BListElt (ListEquations, i);
    VarI = GnlFunctionVar (FunctionI);
    if (GnlVarDir (VarI) != GNL_VAR_LOCAL)
        if (BListAddElt (NewList, (int)GnlFunctionVar (FunctionI)))
            return (GNL_MEMORY_FULL);
}
SetGnlNbOut (*NewGnl, BListSize (NewList));

if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlLocals ((*NewGnl), NewList);

for (i=0; i<BListSize (ListEquations); i++)
{
    FunctionI = (GNL_FUNCTION)BListElt (ListEquations, i);
    VarI = GnlFunctionVar (FunctionI);
    if (GnlVarDir (VarI) == GNL_VAR_LOCAL)
        if (BListAddElt (NewList, (int)GnlFunctionVar (FunctionI)))
            return (GNL_MEMORY_FULL);
}
SetGnlNbLocal (*NewGnl, BListSize (NewList));

if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
SetGnlFunctions ((*NewGnl), NewList);

for (i=0; i<BListSize (ListEquations); i++)
{

```

```

FunctionI = (GNL_FUNCTION)BListElt (ListEquations, i);
VarI = GnlFunctionVar (FunctionI);
if (BListAddElt (NewList, (int)VarI))
    return (GNL_MEMORY_FULL);
}

BListQuickDelete (&ListEquations);

/* We give the clocks of the original 'Gnl'. */  

SetGnlClocks ((*NewGnl), GnlClocks (Gnl));

/* We give the components of the original 'Gnl'. */  

SetGnlComponents ((*NewGnl), GnlComponents (Gnl));

return (GNL_OK);

}

/*-----*/  

/* GnlMergeGnl */  

/*-----*/  

/* This procedure merges a list of Gnls which have been previously */  

/* partitioned by the procedure 'GnlPartitionGnl' below. */  

/* The list of Gnls 'ListGnls' is then delete and the Gnls sons as well */  

/*-----*/  

#define OPTI_CODE
GNL_STATUS GnlMergeGnl (ListGnls, NewGnl)
    BLIST ListGnls;
    GNL *NewGnl;
{
    int i;
    GNL GnI;
    int j;
    GNL_NODE SegmentJ;
    GNL_VAR VarI;
    BLIST NewList;

GnI = (GNL)BListElt (ListGnls, 0);

if ((*NewGnl = (GNL)GNL_CALLOC (1, sizeof(GNL_REC))) == NULL)
    return (GNL_MEMORY_FULL);

/* We affect pointers of the original 'Gnl'. */  

SetGnlName (*NewGnl, GnlName (GnI));
SetGnlSourceFileName (*NewGnl, GnlSourceFileName (GnI));
SetGnlListSourceFiles (*NewGnl, GnlListSourceFiles (GnI));
SetGnlListPorts (*NewGnl, GnlListPorts (GnI));
SetGnlHashNames (*NewGnl, GnlHashNames (GnI));
SetGnlClocks ((*NewGnl), GnlClocks (GnI));
SetGnlComponents ((*NewGnl), GnlComponents (GnI));

/* we create a hash list of list of GNL_PATH_COMPONENT */  

if (BListCreateWithSize (HASH_LIST_PATH_COMPO_SIZE, &NewList))
    return (GNL_MEMORY_FULL);
BSize (NewList) = HASH_LIST_PATH_COMPO_SIZE;

```

gnlpart.c

```
SetGnlListPathComponent ((*NewGnl), NewList);

/* Creating List of nodes segments */  
/* This is the merge of all the segments of all the Gnl's. */  
if (BListCreate (&NewList))  
    return (GNL_MEMORY_FULL);  
for (i=0; i<BListSize (ListGnls); i++)  
{  
    GnlI = (GNL)BListElt (ListGnls, i);  
    for (j=0; j<BListSize (GnlNodesSegments(GnlI)); j++)  
    {  
        SegmentJ = (GNL_NODE)BListElt (GnlNodesSegments(GnlI), j);  
        if (BListAddElt (NewList, (int)SegmentJ))  
            return (GNL_MEMORY_FULL);  
    }  
}  
SetGnlNodesSegments ((*NewGnl), NewList);  
SetGnlFirstNode ((*NewGnl), NULL); /* Next created node will invoke*/  
SetGnlLastNode ((*NewGnl), NULL); /* a new segment. */  
  
#ifdef OPTI_CODE  
    for (i=0; i<BListSize (ListGnls); i++)  
    {  
        GnlI = (GNL)BListElt (ListGnls, i);  
        for (j=0; j<BListSize (GnlLocals(GnlI)); j++)  
        {  
            VarI = (GNL_VAR)BListElt (GnlLocals(GnlI), j);  
            SetGnlVarHook (VarI, NULL);  
        }  
    }  
#endif  
  
/* Creating the list of locals */  
if (BListCreate (&NewList))  
    return (GNL_MEMORY_FULL);  
for (i=0; i<BListSize (ListGnls); i++)  
{  
    GnlI = (GNL)BListElt (ListGnls, i);  
    for (j=0; j<BListSize (GnlLocals(GnlI)); j++)  
    {  
        VarI = (GNL_VAR)BListElt (GnlLocals(GnlI), j);  
  
#ifdef OPTI_CODE  
        if (!GnlVarHook (VarI))  
        {  
            SetGnlVarHook (VarI, 1);  
            if (BListAddElt (NewList, (int)VarI))  
                return (GNL_MEMORY_FULL);  
        }  
#else  
        if (!BListMemberOfList (NewList, VarI, IntIdentical))  
            if (BListAddElt (NewList, (int)VarI))  
                return (GNL_MEMORY_FULL);  
#endif  
    }  
}
```

gnlpart.c

```
SetGnlLocals((*NewGnl), NewList);
SetGnlNbLocal (*NewGnl, BListSize (NewList));

#ifndef OPTI_CODE
for (i=0; i<BListSize (ListGnls); i++)
{
    GnlI = (GNL)BListElt (ListGnls, i);
    for (j=0; j<BListSize (GnlInputs(GnlI)); j++)
    {
        VarI = (GNL_VAR)BListElt (GnlInputs(GnlI), j);
        SetGnlVarHook (VarI, NULL);
    }
}
#endif

/* Creating the list of inputs */
if (BListCreate (&NewList))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (ListGnls); i++)
{
    GnlI = (GNL)BListElt (ListGnls, i);
    for (j=0; j<BListSize (GnlInputs(GnlI)); j++)
    {
        VarI = (GNL_VAR)BListElt (GnlInputs(GnlI), j);
        #ifdef OPTI_CODE
            if (!GnlVarHook (VarI) &&
                ((GnlVarDir (VarI) == GNL_VAR_INPUT) ||
                 (GnlVarDir (VarI) == GNL_VAR_INOUT)))
            {
                SetGnlVarHook (VarI, 1);
                if (BListAddElt (NewList, (int)VarI))
                    return (GNL_MEMORY_FULL);
            }
        #else
            if (((GnlVarDir (VarI) == GNL_VAR_INPUT) ||
                 (GnlVarDir (VarI) == GNL_VAR_INOUT)) &&
                !BListMemberOfList (NewList, VarI, IntIdentical))
                if (BListAddElt (NewList, (int)VarI))
                    return (GNL_MEMORY_FULL);
        #endif
    }
}
SetGnlInputs (*NewGnl, NewList);
SetGnlNbIn (*NewGnl, BListSize (NewList));

#ifndef OPTI_CODE
for (i=0; i<BListSize (ListGnls); i++)
{
    GnlI = (GNL)BListElt (ListGnls, i);
    for (j=0; j<BListSize (GnlOutputs(GnlI)); j++)
    {
        VarI = (GNL_VAR)BListElt (GnlOutputs(GnlI), j);
        SetGnlVarHook (VarI, NULL);
    }
}
#endif
```

```

/* Creating the list of outputs */  

if (BListCreate (&NewList))  

    return (GNL_MEMORY_FULL);  

for (i=0; i<BListSize (ListGnls); i++)  

{
    GnI = (GNL)BListElt (ListGnls, i);
    for (j=0; j<BListSize (GnlOutputs (GnI)); j++)
    {
        VarI = (GNL_VAR)BListElt (GnlOutputs (GnI), j);
        #ifdef OPTI_CODE
            if (!GnlVarHook (VarI) &&
                ((GnlVarDir (VarI) == GNL_VAR_OUTPUT) ||
                 (GnlVarDir (VarI) == GNL_VAR_INOUT)))
            {
                SetGnlVarHook (VarI, 1);
                if (BListAddElt (NewList, (int)VarI))
                    return (GNL_MEMORY_FULL);
            }
        #else
            if (((GnlVarDir (VarI) == GNL_VAR_OUTPUT) ||
                  (GnlVarDir (VarI) == GNL_VAR_INOUT)) &&
                !BListMemberOfList (NewList, VarI, IntIdentical))
                if (BListAddElt (NewList, (int)VarI))
                    return (GNL_MEMORY_FULL);
        #endif
        }
    }
SetGnlOutputs ((*NewGnl), NewList);
SetGnlNbOut (*NewGnl, BListSize (NewList));

/* Creating the list of Functions */  

if (BListCreate (&NewList))  

    return (GNL_MEMORY_FULL);  

for (i=0; i<BListSize (ListGnls); i++)  

{
    GnI = (GNL)BListElt (ListGnls, i);

    for (j=0; j<BListSize (GnlFunctions (GnI)); j++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (GnI), j);
        if (BListAddElt (NewList, (int)VarI))
            return (GNL_MEMORY_FULL);
    }
}
SetGnlFunctions ((*NewGnl), NewList);

/* Freeing the partitions */  

for (i=0; i<BListSize (ListGnls); i++)
{
    GnI = (GNL)BListElt (ListGnls, i);
    BListQuickDelete (&GnlInputs (GnI));
    BListQuickDelete (&GnlOutputs (GnI));
    BListQuickDelete (&GnlLocals (GnI));
}

```

gnlpart.c

```
BListQuickDelete (&GnlFunctions (GnlI));
BListQuickDelete (&GnlNodesSegments (GnlI));
free ((char*)GnlI);
}

BListQuickDelete (&ListGnls);

GnlResetVarHook (*NewGnl);

    return (GNL_OK);
}

/*-----
/* GnlBuildDadsOnNode
/*-----
/* This procedure builds all the Dads of each Nodes from the Node 'Node'*/
/*-----*/
GNL_STATUS GnlBuildDadsOnNode (Node, Father)
    GNL_NODE Node;
    GNL_NODE Father;
{
    int                 i;
    GNL_VAR             Var;
    BLIST               NewList;
    GNL_NODE            NodeI;

    SetGnlNodeHook (Node, NULL);

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR) GnlNodeSons (Node);
        if (!GnlVarDads (Var))
        {
            if (BListCreateWithSize (1, &NewList))
                return (GNL_MEMORY_FULL);
            SetGnlVarDads (Var, NewList);
        }
        NewList = GnlVarDads (Var);
        if (BListAddElt (NewList, (int)Node))
            return (GNL_MEMORY_FULL);

        if (!GnlNodeDads (Node))
        {
            if (BListCreateWithSize (1, &NewList))
                return (GNL_MEMORY_FULL);
            SetGnlNodeDads (Node, NewList);
        }

        NewList = (BLIST) GnlNodeDads (Node);
        if (BListAddElt (NewList, (int)Father))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
```

gnlpart.c

```
}

if (!GnlNodeDads (Node))
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlNodeDads (Node, NewList);
}

NewList = (BLIST)GnlNodeDads (Node);
if (BListAddElt (NewList, (int)Father))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    if (GnlBuildDadsOnNode (NodeI, Node))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlAssignUpPartitionIdOnNode
*/
/*-----*/
/* This procedure tags up recursively each GNL_NODE 'Node' with the tag */
/* 'PartId'. First we tag down this node with the tag 'PartId' then we */
/* tag it up. */
/*-----*/
void GnlTagDownPartitionIdOnNode ();
void GnlAssignUpPartitionIdOnNode (Node, PartId)
    GNL_NODE Node;
    int          PartId;
{
    BLIST      ListDads;
    int         i;
    GNL_VAR    VarI;
    GNL_NODE   NodeI;

GnlTagDownPartitionIdOnNode (Node, PartId);

ListDads = GnlNodeDads (Node);

for (i=0; i<BListSize (ListDads); i++)
{
    /* If the Dad of this node is a GNL_VAR then we continue */
    VarI = (GNL_VAR)BListElt (ListDads, i);
    if (GnlVarIsVar (VarI))
        continue;
    else
    {
        NodeI = (GNL_NODE)BListElt (ListDads, i);
        GnlAssignUpPartitionIdOnNode (NodeI, PartId);
    }
}
```

```

}

/*-----*/
/* GnlAssignUpPartitionId */
/*-----*/
/* This procedure tags up all the nodes recursively which are the Dads */
/* of 'Var'. */
/*-----*/
void GnlAssignUpPartitionId (Var, PartId)
    GNL_VAR Var;
    int PartId;
{
    BLIST ListDads;
    int i;
    GNL_NODE NodeI;

    ListDads = GnlVarDads (Var);

    for (i=0; i<BListSize (ListDads); i++)
    {
        NodeI = (GNL_NODE)BListElt (ListDads, i);
        GnlAssignUpPartitionIdOnNode (NodeI, PartId);
    }
}

/*-----*/
/* GnlTagDownPartitionIdOnNode */
/*-----*/
/* This procedure tags the node 'Node' with the value 'PartId' and goes */
/* recursively. If the node has already the Tag 'PartId' then we can */
/* return because we know we already processed it. If we reach a Var */
/* then we propagate the Tag up to all the nodes using this var and we */
/* do this recursively. */
/*-----*/
void GnlTagDownPartitionIdOnNode (Node, PartId)
    GNL_NODE Node;
    int PartId;
{
    GNL_VAR Var;
    int i;
    GNL_NODE NodeI;

    /* already visited. */
    if (GnlNodeHook (Node) == (void*)PartId)
        return;

    SetGnlNodeHook (Node, (void*)PartId);

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
}

```

```

Var = (GNL_VAR)GnlNodeSons (Node);
if ((GnlVarHook (Var) != (void*)0) &&
    (GnlVarHook (Var) != (void*)PartId))
{
    fprintf (stderr, " ERROR: during partitioning logic\n");
    exit (0);
}

/* If Var tagged with same value then we already passed thru it */
if (GnlVarHook (Var) == (void*)PartId)
    return;

SetGnlVarHook (Var, PartId);

/* We tag up recursively all the Nodes using this var.          */
GnlAssignUpPartitionId (Var, PartId);
return;
}

/* We propagate down the tag on each node.                      */
for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    GnlTagDownPartitionIdOnNode (NodeI, PartId);
}

return ;
}

/*
 * GnlTagDownPartitionIdOnVar
 */
/*
 * This procedure tags all the logic expression from the variable 'Var' */
/* by the value 'PartId'.                                         */
/*
void GnlTagDownPartitionIdOnVar (Var, PartId)
    GNL_VAR Var;
    int      PartId;
{
    GNL_NODE Node;
    GNL_FUNCTION Function;

    Function = GnlVarFunction (Var);
    if (!Function)
        return;

    Node = GnlFunctionOnSet (Function);

    GnlTagDownPartitionIdOnNode (Node, PartId);
}

/*
 * GnlResetDadsInNode
 */
/*
void GnlResetDadsInNode (Node)
    GNL_NODE Node;

```

gnlpart.c

```
{  
    int             i;  
    GNL_NODE NodeI;  
    GNL_VAR Var;  
  
    SetGnlNodeDads (Node, NULL);  
    SetGnlNodeHook (Node, 0);  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
        return;  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR)GnlNodeSons (Node);  
        SetGnlVarDads (Var, NULL);  
        SetGnlVarHook (Var, NULL);  
        return;  
    }  
  
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
    {  
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);  
        GnlResetDadsInNode (NodeI);  
    }  
}  
  
/*-----*/  
/* GnlCleanDadsInNode */  
/*-----*/  
/*-----*/  
/* This procedure delete recursively from the node 'Node' all the Dads */  
/* lists of the Nodes and Vars. */  
/*-----*/  
void GnlCleanDadsInNode (Node)  
{  
    GNL_NODE Node;  
  
    SetGnlNodeHook (Node, NULL);  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
        return;  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR)GnlNodeSons (Node);  
        if (GnlVarDads (Var))  
        {  
            BListQuickDelete (&(GnlVarDads (Var)));  
            SetGnlVarDads (Var, NULL);  
        }  
  
        if (GnlNodeDads (Node))  
        {  
    }
```

gnlpart.c

```
        BListQuickDelete (&(GnlNodeDads (Node)));
        SetGnlNodeDads (Node, NULL);
    }

    return;
}

if (GnlNodeDads (Node))
{
    BListQuickDelete (&(GnlNodeDads (Node)));
    SetGnlNodeDads (Node, NULL);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
    GnlCleanDadsInNode (NodeI);
}
}

/*-----*/
/* GnlCleanNodeAndVars */
/*-----*/
/* This procedure removes the lists that we have created in each Node */
/* and Var starting from the GnlFunctions. */
/*-----*/
void GnlCleanNodeAndVars (Gnl)
{
    GNL      Gnl;
    {
        int          i;
        GNL_FUNCTION FunctionI;
        GNL_VAR     VarI;

        for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
        {
            VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
            FunctionI = GnlVarFunction (VarI);
            GnlCleanDadsInNode (GnlFunctionOnSet (FunctionI));
        }
    }

    /*-----*/
    /* GnlPartitionGnlQuick */
    /*-----*/
    /* This procedure partitions a 'Gnl' into a list of GNL. All equations */
    /* having transitive common variables are put together in the same */
    /* partition. Ex: f1 = a+b, f2 = a+c, f3 = b+i, f4 = z+x; */
    /* We will get two partitions: */
    /*      Partition 1 = {f1 = a+b, f2 = a+c, f3 = b+i} */
    /*      Partition 2 = {f4 = z+x;} */
    /*-----*/
    GNL_STATUS GnlPartitionGnlQuick (Gnl, ListGnls)
    {
        GNL      Gnl;
        BLIST   *ListGnls;
    }
}
```

```

int          i;
int          j;
GNL_VAR      VarI;
GNL_VAR      VarJ;
GNL_FUNCTION FunctionI;
BLIST        SupportI;
BLIST        MergedListOfSupports;
BLIST        ListOfListEquations;
BLIST        ListEquationsI;
GNL          NewGnl;
int          PartId;
GNL_NODE     NodeI;
BLIST        NewList;
BLIST        HashTableNames;
BLIST        BucketI;
BLIST        Support;
BLIST        ListEquations;

if (BListCreate (ListGnls))
    return (GNL_MEMORY_FULL);

/* We reset the Hook and Dads field of all the variables. */
GnlResetVarHook (Gnl);
GnlResetVarDads (Gnl);

/* We first scan the GNL_NODE nodes to reset the 'Dads' field. */
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    GnlResetDadsInNode (GnlFunctionOnSet (FunctionI));
}

/* For all the functions we compute the Dads of each node and dads
   of each variables. */
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (!FunctionI)
        continue;

    if (GnlBuildDadsOnNode (GnlFunctionOnSet (FunctionI), VarI))
        return (GNL_MEMORY_FULL);
}

PartId = 0;
for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    fprintf (stderr, "%c Merging [%d/%d]", 13, i,
             BListSize (GnlFunctions (Gnl)));
    fflush (stderr);
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
}

```

```

/* The variable has already an assigned partition. */  

NodeI = GnlFunctionOnSet (FunctionI);  

if (GnlNodeHook (NodeI))  

    continue;  

PartId++;  

/* We tag all the low recursive expression from the variable */  

/* 'VarI' by the value 'PartId'. */  

GnlTagDownPartitionIdOnVar (VarI, PartId);  

}  

fprintf (stderr, "%c Merging [%d/%d]\n", 13, i,  

        BListSize (GnlFunctions (Gnl)));  

fflush (stderr);  

if (BListCreateWithSize (PartId, &MergedListOfSupports))  

    return (GNL_MEMORY_FULL);  

if (BListCreateWithSize (PartId, &ListOfListEquations))  

    return (GNL_MEMORY_FULL);  

for (i=0; i<PartId; i++)  

{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (MergedListOfSupports, (int)NewList))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListOfListEquations, (int)NewList))
        return (GNL_MEMORY_FULL);
}
  

/* Building the supports of each partition. */  

HashTableNames = GnlHashNames (Gnl);  

for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        PartId = (int)GnlVarHook (VarJ);
        if (!PartId)
            continue;
        Support = (BLIST)BListElt (MergedListOfSupports, PartId-1);
        if (BListAddElt (Support, (int)VarJ))
            return (GNL_MEMORY_FULL);
    }
}
  

/* Placing each equation in the corresponding partition. */  

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    NodeI = GnlFunctionOnSet (FunctionI);
    PartId = (int)GnlNodeHook (NodeI);
    ListEquations = (BLIST)BListElt (ListOfListEquations, PartId-1);
    if (BListAddElt (ListEquations, (int)FunctionI))

```

gnlpart.c

```
        return (GNL_MEMORY_FULL);
    }

/* We remove the lists that we have created in each Nodes and Vars      */
/* starting from the GnlFunctions.                                         */
GnlCleanNodeAndVars (Gnl);

for (i=0; i<BListSize (ListOfListEquations); i++)
{
    fprintf (stderr, "%c Creating Gnl [%d/%d]", 13, i,
             BListSize (ListOfListEquations));
    fflush (stderr);
    ListEquationsI = (BLIST)BListElt (ListOfListEquations, i);
    SupportI = (BLIST)BListElt (MergedListOfSupports, i);

    if (GnlCreateGnlFromFunction (Gnl, ListEquationsI, SupportI,
                                   &NewGnl))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (*ListGnls, (int)NewGnl))
        return (GNL_MEMORY_FULL);
}

fprintf (stderr, "%c Creating Gnl [%d/%d]\n", 13, i,
         BListSize (ListOfListEquations));
fflush (stderr);

return (GNL_OK);
}

-----*/
/* GnlPartitionGnl                                         */
/*-----*/
/* This procedure partitions a 'Gnl' into a list of GNL.           */
/*-----*/
GNL_STATUS GnlPartitionGnl (Gnl, ListGnls)
    GNL          Gnl;
    BLIST       *ListGnls;
{
    int           i;
    GNL_VAR     Vari;
    GNL_FUNCTION FunctionI;
    BLIST       SupportI;
    BLIST       ListSupports;
    BLIST       ListFunctions;
    BLIST       MergedListOfSupports;
    BLIST       ListOfListEquations;
    BLIST       ListEquationsI;
    GNL          NewGnl;

    if (BListCreate (ListGnls))
        return (GNL_MEMORY_FULL);

    if (BListCreate (&ListSupports))
        return (GNL_MEMORY_FULL);
```

gnlpart.c

```
if (BListCreate (&ListFunctions))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);

    if (BListAddElt (ListFunctions, (int)FunctionI))
        return (GNL_MEMORY_FULL);

    if (GnlGetFunctionSupport (GnlFunctionOnSet (FunctionI),
                               &SupportI))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListSupports, (int)SupportI))
        return (GNL_MEMORY_FULL);
}

/* We reset the Hook field of all the variables. */  

GnlResetVarHook (Gnl);

if (GnlMergeFunctions (ListSupports, ListFunctions,
                      &MergedListOfSupports, &ListOfListEquations))
    return (GNL_MEMORY_FULL);

BListDelete (&ListSupports, BListQuickDelete);
BListQuickDelete (&ListFunctions);

for (i=0; i<BListSize (ListOfListEquations); i++)
{
    ListEquationsI = (BLIST)BListElt (ListOfListEquations, i);
    SupportI = (BLIST)BListElt (MergedListOfSupports, i);

    if (GnlCreateGnlFromFunction (Gnl, ListEquationsI, SupportI,
                                 &NewGnl))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (*ListGnls, (int)NewGnl))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}/*-----*/
```

gnlposopt.c

```
/*-----*/  
/*-----*/  
/*      File:          gnlpostopt.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Description: */  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY           /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "gnlopt.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnloption.h"  
#include "gnlestim.h"  
#include "time.h"  
  
#include "blist.e"  
#include "timecomp.e"  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern GNL_ENV    G_GnlEnv;  
extern GNL_VAR    GnlGetOriginalVar ();  
  
/*-----*/  
/* Global variables. */  
/*-----*/  
  
/*-----*/  
/* GnlPrintMotherCellInterface */  
/*-----*/  
void GnlPrintMotherCellInterface (MotherCell)  
    LIB_CELL MotherCell;  
  
{  
    int             i;  
    GNL_VAR InputOrigin;  
    GNL_VAR Input;  
  
    printf (" MOTHER CELL = %s\n", LibHCellName (MotherCell));  
    for (i=0; i<BListSize (LibHCellInputsOrigin (MotherCell)); i++)
```

gnlposopt.c

```
{  
    InputOrigin = (GNL_VAR)BListElt (  
        LibHCellInputsOrigin (MotherCell), i);  
    Input = (GNL_VAR)BListElt (LibHCellInputs (MotherCell), i);  
    printf ("%s <-> %s\n", GnlVarName (InputOrigin),  
        GnlVarName (Input));  
}  
printf ("\n");  
}  
  
/*-----*/  
/* GnlGetNewFormalName */  
/*-----*/  
char *GnlGetNewFormalName (FormalName, CurrentDeriveCell, NewDeriveCell)  
{  
    char *FormalName;  
    LIB_DERIVE_CELL CurrentDeriveCell;  
    LIB_DERIVE_CELL NewDeriveCell;  
    int Index;  
    int i;  
    GNL_VAR InputOrigin;  
    GNL_VAR NormeInput;  
    LIB_CELL CurrentMotherCell;  
    LIB_CELL NewMotherCell;  
    GNL_VAR NewFormalVar;  
  
    Index = -1;  
    CurrentMotherCell = LibDeriveCellMotherCell (CurrentDeriveCell);  
    for (i=0; i<BListSize (LibHCellInputsOrigin (CurrentMotherCell)); i++)  
    {  
        InputOrigin = (GNL_VAR)BListElt (  
            LibHCellInputsOrigin (CurrentMotherCell), i);  
        if (!strcmp (FormalName, GnlVarName (InputOrigin)))  
        {  
            Index = i;  
            break;  
        }  
    }  
    if (Index == -1)  
    {  
        fprintf (stderr, " ERROR: in replacing cell procedure\n");  
        exit (1);  
    }  
  
    NormeInput = (GNL_VAR)BListElt (LibHCellInputs (CurrentDeriveCell),  
        Index);  
  
    NewMotherCell = LibDeriveCellMotherCell (NewDeriveCell);  
    NewFormalVar = GnlGetOriginalVar (NewDeriveCell, NewMotherCell,  
        NormeInput);  
  
    return (GnlVarName (NewFormalVar));  
}  
/*-----*/
```

gnlposopt.c

```
/* GnlFindCellsUserCompo */  
/*-----*/  
/* Extracts the mother cell LIB_CELL and derive cell LIB_DERIVE_CELL of */  
/* the user compo 'UserCompo'. */  
/*-----*/  
void GnlFindCellsUserCompo (UserCompo, MotherCell, DeriveCell)  
    GNL_USER_COMPONENT UserCompo;  
    LIB_CELL *MotherCell;  
    LIB_DERIVE_CELL *DeriveCell;  
{  
    BLIST ListEquivCells;  
    int i;  
    LIB_CELL MotherCellI;  
    LIB_DERIVE_CELL DeriveCellI;  
  
    ListEquivCells = GnlUserComponentEquivCells (UserCompo);  
    *MotherCell = NULL;  
    *DeriveCell = NULL;  
  
    for (i=0; i<BListSize (ListEquivCells); i++)  
    {  
        DeriveCellI = (LIB_DERIVE_CELL)BListElt (ListEquivCells, i);  
        MotherCellI = LibDeriveCellMotherCell (DeriveCellI);  
        if (!strcmp (GnlUserComponentName (UserCompo),  
                    LibHCellName (MotherCellI)))  
        {  
            *MotherCell = MotherCellI;  
            *DeriveCell = DeriveCellI;  
            break;  
        }  
    }  
}  
  
/*-----*/  
/* GnlReplaceUserCompoByDeriveCell */  
/*-----*/  
/* This procedure realizes the physical replacement of the current cell */  
/* 'UserCompo' by the selected derive cell 'NewDeriveCell'. The two */  
/* cells involved have the same functionality but modulo the polarity. */  
/*-----*/  
/* For now, the functionality of the two cells must be exactly the same */  
/* (e.g. same Bdd). */  
/* We need to implement the case where the two Bdds are complemented so */  
/* we have to insert an inverter which is 'BestInverter'. */  
/*-----*/  
GNL_STATUS GnlReplaceUserCompoByDeriveCell (Gnl, UserCompo, NewDeriveCell,  
                                             BestInverter,  
                                             ListInverterCells)  
{  
    GNL Gnl;  
    GNL_USER_COMPONENT UserCompo;  
    LIB_DERIVE_CELL NewDeriveCell;  
    LIB_DERIVE_CELL BestInverter;  
    BLIST ListInverterCells;  
    LIB_CELL NewMotherCell;
```

gnlposopt.c

```
LIB_CELL           CurrentMotherCell;
LIB_CELL           InverterMotherCell;
BLIST              ListEquivCells;
BLIST              CellInputs;
GNL_VAR            VarI;
int                i;
LIBC_CELL          NewLibcCell;
LIB_DERIVE_CELL   CurrentDeriveCell;
GNL_ASSOC          AssocI;
char               *FormalI;
char               *NewFormalI;
GNL_VAR            ActualI;
BLIST              CompoInterface;
BLIST              Interface;
char               *InstanceName;
GNL_USER_COMPONENT NewUserCompo;
GNL_ASSOC          NewAssoc;
GNL_ASSOC          AssocOut;
BLIST              InverterInputsOrigin;
GNL_VAR            InputFormal;
GNL_VAR            NewVar;
GNL_VAR_TRAVERSAL_INFO NewTraversalInfo;

/* There was no cell associated before so the replacement is not valid*/
if (!GnlUserComponentCellDef (UserCompo))
    return (GNL_OK);

/* We look for the current Mother Cell corresponding to the current */
/* user component and its associated derive cell.                      */
ListEquivCells = GnlUserComponentEquivCells (UserCompo);
GnlFindCellsUserCompo (UserCompo, &CurrentMotherCell,
                       &CurrentDeriveCell);

if (!CurrentMotherCell)
{
    /* Oops ! we were not able to find the associated mother cell */
    return (GNL_OK);
}

/* Actually we ask to replace the current by itself.                  */
if (CurrentDeriveCell == NewDeriveCell)
{
    return (GNL_OK);
}

if (GetBddPtr (LibDeriveCellBdd (CurrentDeriveCell)) != 
    GetBddPtr (LibDeriveCellBdd (NewDeriveCell)))
{
    /* Common: the two functionalities are different. No way to      */
    /* make the replacement.                                         */
    return (GNL_OK);
}

/* If the two Cells have complemented functionality we have to insert*/
/* an inverter.                                                 */
*/
```

```

if ((LibDeriveCellBdd (CurrentDeriveCell) !=
     LibDeriveCellBdd (NewDeriveCell)))
{
    /* For now we do not consider complemented functionalities      */
    /* between the 'CurrentDeriveCell' and the new 'NewDeriveCell'.  */
    return (GNL_OK);

/* we look for the assoc output of the user component           */
AssocOut = NULL;
CompoInterface = GnlUserComponentInterface (UserCompo);
for (i=0; i<BListSize (CompoInterface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (CompoInterface, i);
    FormalI = (char*)GnlAssocFormalPort (AssocI);
    ActualI = GnlAssocActualPort (AssocI);

    if (!strcmp (FormalI, LibHCellOutput (CurrentMotherCell)))
    {
        AssocOut = AssocI;
        break;
    }
}

/* We create a new user component for the inverter.          */
InverterMotherCell = LibDeriveCellMotherCell (BestInverter);

if (BListCreateWithSize (1, &Interface))
    return (GNL_MEMORY_FULL);

SetGnlInstanceId (Gnl, GnlInstanceId (Gnl)+1);
if (GnlStrAppendIntCopy ("m", GnlInstanceId (Gnl), &InstanceName))
    return (GNL_MEMORY_FULL);

if (GnlCreateUserComponent (LibHCellName (InverterMotherCell),
                           InstanceName, NULL, Interface,
                           &NewUserCompo))
    return (GNL_MEMORY_FULL);

/* The output actual net of the Inverter is the actual output   */
/* of the user component.                                         */
if (GnlCreateAssoc (&NewAssoc))
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort (NewAssoc,
                      LibHCellOutput (InverterMotherCell));
SetGnlAssocActualPort (NewAssoc, GnlAssocActualPort (AssocOut));
if (BListAddElt (Interface, (int)NewAssoc))
    return (GNL_MEMORY_FULL);

/* We create a new var to make the net between the inverter and  */
/* new cell that we will put.                                     */
if (GnlCreateUniqueVar (Gnl, "RS", &NewVar))
    return (GNL_MEMORY_FULL);
if (BListAddElt (GnlLocals (Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);
SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);
SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);

```

```

if (GnlCreateVarTraversalInfo (&NewTraversalInfo))
    return (GNL_MEMORY_FULL);
/* we hook the field with this new structure. */ 
SetGnlVarHook (NewVar, NewTraversalInfo);
if (GnlVarTraversalInfoAddAssoc (NewVar, AssocOut))
    return (GNL_MEMORY_FULL);

/* The actual output of the current user compo is replaced by */
/* this var. */ 
SetGnlAssocActualPort (AssocOut, NewVar);

/* we create the association of the input of the inverter */
if (GnlCreateAssoc (&NewAssoc))
    return (GNL_MEMORY_FULL);
SetGnlAssocActualPort (NewAssoc, NewVar);
InverterInputsOrigin = LibHCellInputsOrigin (CurrentMotherCell);
InputFormal = (GNL_VAR)BListElt (InverterInputsOrigin, 0);
SetGnlAssocFormalPort (NewAssoc, GnlVarName (InputFormal));
if (BListAddElt (Interface, (int)NewAssoc))
    return (GNL_MEMORY_FULL);
if (GnlVarTraversalInfoAddAssoc (NewVar, NewVar))
    return (GNL_MEMORY_FULL);
SetGnlUserComponentInterface (NewUserCompo, Interface);

/* we link the component with its corresponding LIBC cell */
SetGnlUserComponentCellDef (NewUserCompo,
                            LibHCellLibcCell (InverterMotherCell));
SetGnlUserComponentEquivCells (NewUserCompo, ListInverterCells);

if (BListAddElt (GnlComponents (Gnl), (int)NewUserCompo))
    return (GNL_MEMORY_FULL);

}

/* Now we get the New mother cell of the cell we will put. */
NewMotherCell = LibDeriveCellMotherCell (NewDeriveCell);
NewLibcCell = LibHCellLibcCell (NewMotherCell);

if (GnlEnvPrintResizedGates ())
{
    fprintf (stderr, " WARNING: Replacing Cell <%s> by Cell <%s>\n",
            LibHCellName (CurrentMotherCell),
            LibHCellName (NewMotherCell));
}

CompoInterface = GnlUserComponentInterface (UserCompo);
for (i=0; i<BListSize (CompoInterface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (CompoInterface, i);
    FormalI = (char*)GnlAssocFormalPort (AssocI);
    ActualI = GnlAssocActualPort (AssocI);

    /* If we are treating the output of the cell we replace by the */
    /* output of the new mother cell. */ 
    if (!strcmp (FormalI, LibHCellOutput (CurrentMotherCell)))
    {
        SetGnlAssocFormalPort (AssocI,

```

gnlposopt.c

```
                                LibHCellOutput (NewMotherCell));
    continue;
}

NewFormalI = GnlGetNewFormalName (FormalI, CurrentDeriveCell,
                                 NewDeriveCell);

SetGnlAssocFormalPort (AssocI, NewFormalI);
}

SetGnlUserComponentName (UserCompo, LibHCellName (NewMotherCell));
SetGnlUserComponentCellDef (UserCompo, NewLibcCell);

return (GNL_OK);
}

/*-----
/* GnlFindFasterCellTrivial
*/
LIB_DERIVE_CELL GnlFindFasterCellTrivial (UserCompo)
  GNL_USER_COMPONENT UserCompo;
{
    int             i;
    BLIST           ListEquivCells;
    LIB_CELL        CurrentMotherCell;
    LIB_DERIVE_CELL CurrentDeriveCell;
    LIB_DERIVE_CELL DeriveCellI;

    ListEquivCells = GnlUserComponentEquivCells (UserCompo);
    GnlFindCellsUserCompo (UserCompo, &CurrentMotherCell,
                           &CurrentDeriveCell);

    for (i=BListSize (ListEquivCells)-1; i>=0; i--)
    {
        DeriveCellI = (LIB_DERIVE_CELL)BListElt (ListEquivCells, i);
        if (LibDeriveCellBdd (CurrentDeriveCell) ==
            LibDeriveCellBdd (DeriveCellI))
        {
            return (DeriveCellI);
        }
    }
}

return (NULL);
}

/*-----
/* GnlReplaceComponentByDeriveCell
*/
GNL_STATUS GnlReplaceComponentByDeriveCell (Gnl, Component, GnlLib)
  GNL             Gnl;
  GNL_COMPONENT   Component;
  GNL_LIB         GnlLib;
{
  GNL_USER_COMPONENT UserCompo;
```

gnlposopt.c

```
LIB_DERIVE_CELL      BestCell;
BLIST              ListInverters;
LIB_DERIVE_CELL      BestInv;

switch (GnlComponentType (Component)) {

    case GNL_USER_COMPO:
        UserCompo = (GNL_USER_COMPONENT) Component;

        if ((BestCell = GnlFindFasterCellTrivial (UserCompo))
            == NULL)
            return (GNL_OK);

        ListInverters = GnlHLibInverters (GnlLib);
        BestInv = (LIB_DERIVE_CELL) BListElt (ListInverters,
                                              BListSize (ListInverters)-1);

        if (GnlReplaceUserCompoByDeriveCell (Gnl, UserCompo,
                                              BestCell, BestInv, ListInverters))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);

    case GNL_SEQUENTIAL_COMPO:
        return (GNL_OK);

    case GNL_TRISTATE_COMPO:
        return (GNL_OK);

    case GNL_BUF_COMPO:
        return (GNL_OK);

    default:
        return (GNL_OK);
}

/*
 * GnlReplaceCellInGnl
 */
GNL_STATUS GnlReplaceCellInGnl (Gnl, GnlLib)
    GNL          Gnl;
    GNL_LIB     GnlLib;
{
    BLIST          Components;
    int           i;
    GNL_COMPONENT ComponentI;

    Components = GnlComponents (Gnl);

    if (!Components)
        return (GNL_OK);

    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (Components, i);
```

gnlposopt.c

```
if (GnlReplaceComponentByDeriveCell (Gnl, ComponentI, GnlLib))
    return (GNL_MEMORY_FULL);
}

return (GNL_OK);

/*-----*/
/* GnlPostOptimizeNetwork */
/*-----*/
GNL_STATUS GnlPostOptimizeNetwork (Nw, HGnlLib)
    GNL_NETWORK          Nw;
    GNL_LIB              HGnlLib;
{
    GNL                  TopGnl;

TopGnl = GnlNetworkTopGnl (Nw);

if (GnlReplaceCellInGnl (TopGnl, HGnlLib))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----*/
/* GnlPostOptimize */
/*-----*/
GNL_STATUS GnlPostOptimize ()
{
    int                  i;
    GNL                  Gnl;
    GNL                  GnlI;
    BLIST                ListGnls;
    LIBC_LIB              GnlLib;
    GNL_LIB              HGnlLib;
    int                  Done;
    FILE                 *OutFile;
    GNL_STATUS            GnlStatus;
    GNL                  TopGnl;
    GNL_NETWORK           GlobalNetwork;
    char                 *LibSourceFileName;

if (GnlEnvOutput())
{
    if ((OutFile = fopen (GnlEnvOutput(), "w")) == NULL)
    {
        fprintf (stderr, " ERROR: Cannot Open Output File '%s'\n",
                 GnlEnvOutput());
        return (GNL_CANNOT_OPEN_OUTFILE);
    }
    fclose (OutFile);
}
```

gnlposopt.c

```
if ((OutFile = fopen (GnlEnvLib(), "r")) == NULL)
{
    fprintf (stderr, " ERROR: Cannot Open Library File '%s'\n",
             GnlEnvLib());
    return (GNL_CANNOT_OPEN_LIBRARY);
}
fclose (OutFile);
if (GnlEnvInputFormat() == GNL_INPUT_VLG)
{
    fprintf (stderr, "\n Reading Verilog file [%s] ... \n",
             GnlEnvInput());
    if (GnlRead (GnlEnvInput(), &ListGnls))
    {
        fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
                 GnlEnvInput());
        return (GNL_CANNOT_OPEN_INPUTFILE);
    }
    fprintf (stderr, " Verilog Netlist Analyzed\n");
}
else
{
    fprintf (stderr,
             " ERROR: Post-Optimization must be done on Verilog netlist
input\n");
    return (GNL_CANNOT_OPEN_INPUTFILE);
}

/* Sort and prints the list 'ListGnls' by putting first the top      */
/* level modules      */
SortTopLevelModules (ListGnls);
GnlPrintTopLevels (ListGnls);

if (!GnlEnvTop())
{
    GnlI = (GNL)BListElt (ListGnls, 0);
    TopGnl = GnlI;
    fprintf (stderr,
             " WARNING: Top Level Module is [%s] by default\n",
             GnlName (TopGnl));
}
else
{
    /* Taking the top level module from the user      */
    TopGnl = NULL;
    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnlI = (GNL)BListElt (ListGnls, i);
        if (!strcmp (GnlName (GnlI), GnlEnvTop()))
        {
            TopGnl = GnlI;
            break;
        }
    }
}

if (TopGnl == NULL)
```

```

{
    fprintf (stderr, " ERROR: cannot find top-level module [%s]\n",
             GnlEnvTop());
    return (GNL_CANNOT_FIND_TOP_LEVEL);
}

/* we create the Global network. */
if (GnlCreateNetwork (TopGnl, &GlobalNetwork))
    return (GNL_MEMORY_FULL);

/* Reading and building the technology library */
fprintf (stderr, "\n Reading Library [%s] ... \n",
         GnlEnvLib());
if (LibRead (GnlEnvLib(), &GnlLib))
{
    fprintf (stderr, " ERROR: Library analysis aborted\n");
    return (GNL_ERROR_LIBRARY_READING);
}
fprintf (stderr, " Library analyzed\n");

fprintf (stderr, "\n Building library elements... \n\n");
if ((GnlStatus = GnlBuildLib (GnlLib)))
    return (GnlStatus);

/* Checking correctness of the network 'GlobalNetwork'. */
if ((GnlStatus = GnlCheckNetwork (GlobalNetwork)))
    return (GnlStatus);

fprintf (stderr,
        "\n Checking Hierachical interfaces from top module [%s]\n",
        GnlName (TopGnl));

/* We verify the correct connection between user components and */
/* their definitions and update some fields (RefCount). */
if (GnlUpdateNCheckHierarchyInterface (GlobalNetwork, TopGnl,
                                       ListGnls))
    return (GNL_MEMORY_FULL);

/* We link each component of all the Gnls in the network with the */
/* libc cells of same name. */
fprintf (stderr, " Linking Components With Libc Cells\n");
if (GnlLinkLib (GlobalNetwork, TopGnl, GnlLib))
    return (GNL_MEMORY_FULL);
HGNLlib = (GNL_LIB)LibHook (GnlLib);
if (GnlStrCopy (GnlEnvLib(), &LibSourceFileName))
    return (GNL_MEMORY_FULL);
SetGnlHLibSourceFileName (HGNLlib, LibSourceFileName);

/* We modify the original GNL_USER_COMPONENT components according to */
/* to their linking libc cell. */
/* Only not combinatorial GNL_USER_COMPONENT are modified */
if (GnlReplacePredefComponentsWithLinkLibCells (GlobalNetwork, TopGnl,
                                               GnlLib))
    return (GNL_MEMORY_FULL);

/* We invoke the control fanout algorithm. */

```

gnlposopt.c

```
if (GnlEnvRespectLibConstraints ())
{
    if (GnlStatus = GnlMeetLibConstraints (GlobalNetwork, GnlLib))
        return (GnlStatus);
}

/* We invoke the post optimisation on the synthesized design */
if (GnlEnvPostOpt ())
{
    fprintf (stderr, " Performing Post-Optimization...\n");

#ifndef XXX_MODE
    if ((GnlStatus = GnlPostOptimizeNetwork (GlobalNetwork, HGNLlib)))
        return (GnlStatus);
#endif

    if (TimeOptByResizing (GlobalNetwork, GnlLib))
        return (GNL_MEMORY_FULL);
    fprintf (stderr, "\n");
}

if (GnlReportDataForEstimation (GlobalNetwork, GnlLib))
    return (GNL_MEMORY_FULL);

/* Printing the optimized mapped verilog netlist. */
fprintf (stderr, " Printing Verilog Netlist '%s'...\n\n",
        GnlEnvOutput());
if ((GnlStatus = GnlPrintVerilogNetwork (GlobalNetwork, HGNLlib,
                                         GnlEnvOutput())))
    return (GnlStatus);

fprintf (stderr, " Post-Optimization Done ! \n");
fprintf (stderr, " \n");

return (GNL_OK);
}

/* ----- */
```

gnlpower.c

```
/*-----*/
/*
/*      File:          gnlpower.c
/*      Version:        1.1
/*      Modifications: -
/*      Documentation: -
/*
/*      Description:           */
/*
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "gnloption.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlestim.h"
#include "gnlmap.h"
#include "time.h"

#include "blist.e"

#include "libutil.e"
#include "timecomp.e"

/*-----*/
/* EXTERN           */
/*-----*/
extern GNL_ENV          G_GnlEnv;
extern BDD_PTR           GetBddPtrFromBdd ();

/*-----*/
/* FORWARD          */
/*-----*/
GNL_STATUS  GnlBestAreaMapNode ();
GNL_STATUS  GnlBestDepthMapNode ();
void        GnlAddPinCapacitance ();
GNL_STATUS  GnlGetMaxMinDelaysFromNode ();
void        GnlGetMaxTechnologyDepthFromNode ();

/*-----*/
/* GnlPowerEstimate */
/*-----*/
/* This procedure computes the power estimations related to the global */
/* netlist 'GlobalNetwork' using library cells of 'Gnllibc'. */
/*-----*/
```

gnlpower.c

```
/*-----*/
GNL_STATUS GnlPowerEstimate (Nw, GnlLibc)
    GNL_NETWORK    Nw;
    LIBC_LIB GnlLibc;
{
    GNL           TopGnl;
    GNL_STATUS    GnlStatus;

    fprintf (stderr, " Performing Power Analysis...\n");

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    TopGnl = GnlNetworkTopGnl (Nw);

    return (GNL_OK);
}

/*-----*/
/* GnlPowerEstimateAfterSynthesis */
/*-----*/
/* This procedure computes the power estimations related to the global */
/* netlist 'GlobalNetwork' using library cells of 'GnlLibc'. */
/*-----*/
GNL_STATUS GnlPowerEstimateAfterSynthesis (Nw, GnlLibc)
    GNL_NETWORK    Nw;
    LIBC_LIB       GnlLibc;
{
    GNL           TopGnl;
    GNL_STATUS    GnlStatus;

    TopGnl = GnlNetworkTopGnl (Nw);
    if (GnlLinkLib (Nw, TopGnl, GnlLibc))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlPowerEstimate (Nw, GnlLibc)))
        return (GnlStatus);

    return (GNL_OK);
}

/*-----*/
```

gnlprint.c

```
/*-----*/  
/*-----*/  
/*      File:          gnlprint.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Description: */  
/*-----*/  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY           /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
#include "gnloption.h"  
  
#include "blist.e"  
  
/*-----*/  
/* DEFINE */  
/*-----*/  
#define PLIB      /* If defined then verilog output is simplified and */  
/*                 and dedicated for 'plib' library (to do verification) */  
#undef PLIB  
  
#define PRINT_PARAMETERS /* If defined then instance parameters */  
/*                   are printed out. */  
#undef PRINT_PARAMETERS  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern BDD_PTR    GetBddPtrFromBdd ();  
extern GNL_VAR     GnlGetOriginalVar ();  
extern LIB_CELL    GetBestAreaLibInverter ();  
extern GNL_ENV     G_GnlEnv;  
extern double      GnlGetNetworkGateArea ();  
extern GNL_STATUS   GnlGetNetworkNetArea ();  
  
/*-----*/  
/* STATIC GLOBAL VARIABLES */  
/*-----*/  
static int  G_PrintModuleCounter = 0;  
  
/*-----*/
```

gnlprint.c

```

/* GnlPrintMapNode
*/
void GnlPrintMapNode (File, Node)
{
    FILE          *File;
    GNL_NODE      Node;
    GNL_NODE      Son;
    GNL_VAR       Var;
    int           i;

switch (GnlNodeOp (Node)) {
    case GNL_VARIABLE:
        Var = (GNL_VAR)GnlNodeSons (Node);
        if (GnlVarIsVss (Var))
            fprintf (File, "{0}");
        else if (GnlVarIsVdd (Var))
            fprintf (File, "{1}");
        else
            fprintf (File, "%s", GnlVarName (Var));
        return;

    case GNL_NOT:
        if (GnlMapNodeInfoCutVar (Node))
        {
            Var = GnlMapNodeInfoCutVar (Node);
            fprintf (File, "%s", GnlVarName (Var));
            return;
        }
        Son = (GNL_NODE)BListElt (GnlNodeSons (Node), 0);
        fprintf (File, "!(");
        GnlPrintMapNode (File, Son);
        fprintf (File, ")");
        return;

    case GNL_AND:
    case GNL_OR:
    case GNL_NOR:
    case GNL_NAND:
    case GNL_XOR:
    case GNL_XNOR:
        if (GnlMapNodeInfoCutVar (Node))
        {
            Var = GnlMapNodeInfoCutVar (Node);
            fprintf (File, "%s", GnlVarName (Var));
            return;
        }
        fprintf (File, "(");
        for (i=0; i<BListSize (GnlNodeSons (Node))-1; i++)
        {
            Son = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
            GnlPrintMapNode (File, Son);
            fprintf (File, "%s", GnlNameFromOp (GnlNodeOp (Node)));
        }
        Son = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        GnlPrintMapNode (File, Son);
}

```

gnlprint.c

```
    fprintf (File, ")");
    return;

    case GNL_CONSTANTE:
        if (GnlNodeSons (Node) == (BLIST) 1)
            fprintf (File, "{1}");
        else
            fprintf (File, "{0}");
        return;

    case GNL_WIRE:
        Son = (GNL_NODE) GnlNodeSons (Node);
        GnlPrintMapNode (File, Son);
        return;

    default:
        GnlError (9 /* Unknown node */);
        return;
    }
}
```

```
/*
 *-----*/
/* GnlPrintVarName */
/*-----*/
/* This procedure makes a side effect on the name of the variable 'Var' */
/* in order to make it Verilog compatible. */
/*-----*/
char *GnlPrintVarName (Var)
    GNL_VAR Var;
{
    int          L;
    int          i;
    int          BracketCase;

    BracketCase = 0;
    L = strlen (GnlVarName (Var));
    for (i=L-1; i>=0; i--)
    {
        switch (GnlVarName (Var) [i]) {

            case '(':
                if (BracketCase)
                    GnlVarName (Var) [i] = '[';
                else
                    GnlVarName (Var) [i] = '_';
                break;

            case ')':
                if (i == L-1)
                {
                    BracketCase = 1;
                    GnlVarName (Var) [i] = ']';
                }
                else

```

gnlprint.c

```
        GnlVarName(Var)[i] = '_';
        break;

    case '@':
    case '.':
        GnlVarName(Var)[i] = '_';
        break;

    default:
        break;
    }

    return (GnlVarName (Var));
}

/*-----
/* GnlNameEffectiveVar
/*-----*/
char *GnlNameEffectiveVar (Var)
    GNL_VAR Var;
{
    char          FChar;

    if (GnlVarIsVdd (Var))
        return ("1'b1");
    if (GnlVarIsVss (Var))
        return ("1'b0");

    return (GnlPrintVarName (Var));
}

/*-----
/* GnlPrintCell
/*-----*/
GNL_STATUS GnlPrintCell (File, Gnl, OutVar, InstanceId, Cell, Node)
    FILE          *File;
    GNL           Gnl;
    GNL_VAR       OutVar;
    int            InstanceId;
    LIB_DERIVE_CELL Cell;
    GNL_NODE      Node;
{
    int          i;
    int          j;
    LIB_CELL     MotherCell;
    GNL_VAR      VarI;
    GNL_VAR      VarCellII;
    char         *Originalname;
    GNL_NODE     NodeJ;
    BDD          Bdd;
    BLIST         NodeSupport;
    BLIST         LibVarSupport;
    BLIST         ListData;
```

gnlprint.c

```
LIB_INPUT_DATA InputDataI;

MotherCell = LibDeriveCellMotherCell (Cell);
fprintf (File, "%s u%d", LibHCellName (MotherCell), InstanceId);

/* Printing data cell */
#ifndef PRINT_PARAMETERS
fprintf (File, "#(%f, %d, %f, %f, %f) ",
        LibHCellArea (MotherCell),
        GnlMapNodeInfoDepth (Node), GnlMapNodeInfoOutCapa (Node),
        GnlMapNodeInfoMaxDelay (Node), GnlMapNodeInfoMinDelay (Node));
#endif

/* Printing the output. */
fprintf (File, "(");
#ifndef PLIB
fprintf (File, "%s , ", GnlPrintVarName (OutVar));
#else
fprintf (File, ".%s (%s) , ", LibHCellOutput (MotherCell),
        GnlPrintVarName (OutVar));
#endif

NodeSupport = GnlMapNodeInfoNodeSupport (Node);
LibVarSupport = GnlMapNodeInfoLibVarSupport (Node);

/* For each input of the Cell ... */
for (i=0; i<BListSize (LibDeriveCellInputs (Cell))-1; i++)
{
    VarI = (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i);
    for (j=0; j<BListSize (GnlMapNodeInfoLibVarSupport (Node)); j++)
    {
        VarCellII = (GNL_VAR)
            BListElt (GnlMapNodeInfoLibVarSupport (Node), j);
        if (VarI == VarCellII)
            break;
    }
    NodeJ = (GNL_NODE)BListElt (GnlMapNodeInfoNodeSupport (Node), j);
    if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
        VarI = (GNL_VAR)GnlNodeSons (NodeJ);
    else
        VarI = GnlMapNodeInfoCutVar (NodeJ);
    VarCellII = GnlGetOriginalVar (Cell, MotherCell, VarCellII);
#ifndef PLIB
    fprintf (File, ".%s (%s) , ", GnlPrintVarName (VarCellII),
            GnlNameEffectiveVar (VarI));
#else
    fprintf (File, "%s , ", GnlNameEffectiveVar (VarI));
#endif
}

VarI = (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i);
for (j=0; j<BListSize (GnlMapNodeInfoLibVarSupport (Node)); j++)
{
    VarCellII = (GNL_VAR)
        BListElt (GnlMapNodeInfoLibVarSupport (Node), j);
    if (VarI == VarCellII)
```

gnlprint.c

```
        break;
    }
NodeJ = (GNL_NODE)BListElt (GnlMapNodeInfoNodeSupport (Node), j);
if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
    VarI = (GNL_VAR)GnlNodeSons (NodeJ);
else
    VarI = GnlMapNodeInfoCutVar (NodeJ);
VarCellI = GnlGetOriginalVar (Cell, MotherCell, VarCellI);
#ifndef PLIB
    fprintf (File, ".%s (%s );", GnlPrintVarName (VarCellI),
             GnlNameEffectiveVar (VarI));
#else
    fprintf (File, "%s ;", GnlNameEffectiveVar (VarI));
#endif
.
return (GNL_OK);
}

/*
 * GnlPrintCFormatCell
 */
void GnlPrintCFormatCell (File, OutVar, InstanceId, Cell, Node)
    FILE           *File;
    GNL_VAR        OutVar;
    int            InstanceId;
    LIB_DERIVE_CELL Cell;
    GNL_NODE       Node;
{
    int             i;
    int             j;
    LIB_CELL        MotherCell;
    GNL_VAR         VarI;
    GNL_VAR         VarCellI;
    char            *Originalname;
    GNL_NODE        NodeJ;
    BDD             Bdd;
    BLIST           NodeSupport;
    BLIST           LibVarSupport;
    BLIST           ListData;
    LIB_INPUT_DATA  InputDataI;

    MotherCell = LibDeriveCellMotherCell (Cell);
    fprintf (File, ".gate %s ", LibHCellName (MotherCell));

    NodeSupport = GnlMapNodeInfoNodeSupport (Node);
    LibVarSupport = GnlMapNodeInfoLibVarSupport (Node);

    /* For each input of the Cell ...
     */
    for (i=0; i<BListSize (LibDeriveCellInputs (Cell))-1; i++)
    {
        VarI = (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i);
        for (j=0; j<BListSize (GnlMapNodeInfoLibVarSupport (Node)); j++)
        {
            VarCellI = (GNL_VAR)
                BListElt (GnlMapNodeInfoLibVarSupport (Node), j);
```

gnlprint.c

```
if (VarI == VarCellI)
    break;
}
NodeJ = (GNL_NODE)BListElt (GnlMapNodeInfoNodeSupport (Node), j);
if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
    VarI = (GNL_VAR)GnlNodeSons (NodeJ);
else
    VarI = GnlMapNodeInfoCutVar (NodeJ);
    VarCellI = GnlGetOriginalVar (Cell, MotherCell, VarCellI);
fprintf (File, "%s=%s ", GnlVarName (VarCellI),
        GnlVarName (VarI));
}

VarI = (GNL_VAR)BListElt (LibDeriveCellInputs (Cell), i);
for (j=0; j<BListSize (GnlMapNodeInfoLibVarSupport (Node)); j++)
{
    VarCellI = (GNL_VAR)
        BListElt (GnlMapNodeInfoLibVarSupport (Node), j);
    if (VarI == VarCellI)
        break;
}
NodeJ = (GNL_NODE)BListElt (GnlMapNodeInfoNodeSupport (Node), j);
if (GnlNodeOp (NodeJ) == GNL_VARIABLE)
    VarI = (GNL_VAR)GnlNodeSons (NodeJ);
else
    VarI = GnlMapNodeInfoCutVar (NodeJ);
    VarCellI = GnlGetOriginalVar (Cell, MotherCell, VarCellI);
fprintf (File, "%s=%s ", GnlVarName (VarCellI),
        GnlVarName (VarI));

/* Printing the output. */
fprintf (File, "%s=%s ", LibHCellOutput (MotherCell),
        GnlVarName (OutVar));
}

/*
 * GnlPrintMappedGnlNode
 */
GNL_STATUS GnlPrintMappedGnlNode (File, Gnl, BestInv, InstanceId,
                                  OutVar, Node, IndexList)
FILE          *File;
GNL           Gnl;
LIB_CELL      BestInv;
int            *InstanceId;
GNL_VAR       OutVar;
GNL_NODE      Node;
int            *IndexList;
{
    int             i;
    BDD             Bdd;
    BDD_PTR        BddPtr;
    BLIST          ListDeriveCells;
    LIB_DERIVE_CELL BestCell;
    GNL_NODE       Son;
    GNL_VAR        Var;
    GNL_VAR        NotVar;
```

gnlprint.c

```
GNL_VAR           InVar;
float             Area;
BLIST             ListData;
LIB_INPUT_DATA   InputDataI;
int               Depth;

switch (GnlNodeOp (Node)) {
    case GNL_VARIABLE:
        fprintf (File, "    assign %s = ", GnlPrintVarName (OutVar));
        Var = (GNL_VAR) GnlNodeSons (Node);
        if (GnlVarIsVss (Var))
            fprintf (File, "1'b0;");
        else if (GnlVarIsVdd (Var))
            fprintf (File, "1'b1;");
        else
        {
            fprintf (File, "%s ;", GnlPrintVarName (Var));
        }
        return (GNL_OK);

    case GNL_NOT:
    case GNL_AND:
    case GNL_OR:
        Var = GnlMapNodeInfoCutVar (Node);
        if (Var != OutVar)
        {
            fprintf (File, "    assign %s = %s ;", GnlPrintVarName
(OutVar),
                    GnlPrintVarName (Var));
            return (GNL_OK);
        }
        Bdd = GnlMapNodeInfoBestBdd (Node);
        BddPtr = GetBddPtrFromBdd (Bdd);
        if ((Bdd == bdd_one ()) ||
            (Bdd == bdd_zero ()))
        {
            if (Bdd == bdd_one ())
                fprintf (File, "    assign %s = 1'b1;",
                        GnlPrintVarName (OutVar));
            else
                fprintf (File, "    assign %s = 1'b0;",
                        GnlPrintVarName (OutVar));
            return (GNL_OK);
        }
        ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
        GnlGetBestAreaCell (LibBddInfoDeriveCells (BddPtr),
                            Bdd, &BestCell, &Area, &Depth);
        if (LibDeriveCellBdd (BestCell) != Bdd)
        {
            printf ("ERROR: bdd differents\n");
            exit (1);
        }
        if (GnlPrintCell (File, Gnl, OutVar, *InstanceId,
                          BestCell, Node))
            return (GNL_MEMORY_FULL);
```

```

(*InstanceId)++;
return (GNL_OK);

case GNL_CONSTANTE:
    fprintf (File, "    assign %s = ", GnlPrintVarName (OutVar));
    if (GnlNodeSons (Node) == (BLIST) 1)
        fprintf (File, "1'b1;");
    else
        fprintf (File, "1'b0;");
    return (GNL_OK);

default:
    GnlError (9 /* Unknown node */);
    return (GNL_OK);
}

/*
 * GnlPrintCFormatMappedGnlNode
 */
void GnlPrintCFormatMappedGnlNode (File, Gnl, BestInv, InstanceId,
                                  OutVar, Node, IndexList)
FILE          *File;
GNL           Gnl;
LIB_CELL      BestInv;
int           *InstanceId;
GNL_VAR       OutVar;
GNL_NODE     Node;
int           *IndexList;

{
    int             i;
    BDD            Bdd;
    BDD_PTR        BddPtr;
    BLIST          ListDeriveCells;
    LIB_DERIVE_CELL BestCell;
    GNL_NODE      Son;
    GNL_VAR       Var;
    GNL_VAR       NotVar;
    GNL_VAR       InVar;
    float          Area;
    BLIST          ListData;
    LIB_INPUT_DATA InputDataI;
    int            Depth;

switch (GnlNodeOp (Node)) {
    case GNL_VARIABLE:
        fprintf (File, "    assign %s = ", GnlVarName (OutVar));
        Var = (GNL_VAR) GnlNodeSons (Node);
        if (GnlVarIsVss (Var))
            fprintf (File, "1'b0;");
        else if (GnlVarIsVdd (Var))
            fprintf (File, "1'b1;");
        else
        {
            fprintf (File, "%s ;", GnlVarName (Var));
        }
}

```

```

        return;

    case GNL_NOT:
    case GNL_AND:
    case GNL_OR:
        Var = GnlMapNodeInfoCutVar (Node);
        if (Var != OutVar)
        {
            fprintf (File, "      assign %s = %s ;", GnlVarName (OutVar),
                     GnlVarName (Var));
            return;
        }
        Bdd = GnlMapNodeInfoBestBdd (Node);
        BddPtr = GetBddPtrFromBdd (Bdd);
        if ((Bdd == bdd_one ()) ||
            (Bdd == bdd_zero ()))
        {
            if (Bdd == bdd_one ())
                fprintf (File, "      assign %s = 1'b1;",
                         GnlVarName (OutVar));
            else
                fprintf (File, "      assign %s = 1'b0;",
                         GnlVarName (OutVar));
            return;
        }
        ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
        GnlGetBestAreaCell (LibBddInfoDeriveCells (BddPtr),
                            Bdd, &BestCell, &Area, &Depth);
        if (LibDeriveCellBdd (BestCell) != Bdd)
        {
            printf ("ERROR: bdd differents\n");
            exit (1);
        }
        GnlPrintCFormatCell (File, OutVar, *InstanceId, BestCell, Node);
        (*InstanceId)++;
        return;

    case GNL_CONSTANTE:
        fprintf (File, "      assign %s = ", GnlVarName (OutVar));
        if (GnlNodeSons (Node) == (BLIST)1)
            fprintf (File, "1'b1;");
        else
            fprintf (File, "1'b0;");
        return;

    default:
        GnlError (9 /* Unknown node */);
        return;
    }
}

/*
 * GnlPrintVerilogSimpleNwHeader
 */
/*
 * Prints official network header in the verilog ouptut file.
 */
void GnlPrintVerilogSimpleNwHeader (File, Nw, GnlLib)

```

gnlprint.c

```
FILE          *File;
GNL_NETWORK   Nw;
GNL_LIB       GnlLib;
{

    fprintf (File,
              "// ======\n");
    fprintf (File,
              "// This Verilog file has been automatically generated \n");
    fprintf (File,
              "// by %s %s\n", GNL_TOOL_NAME, GNL_MAPIT_VERSION);
    fprintf (File,
              "\n");
    fprintf (File,
              "// Copyright (c) 1994-99 by AVANT! Corp.\n");
    fprintf (File,
              "\n");
    fprintf (File,
              "// Date:");
    GnlPrintDate (File);
    fprintf (File, "\n");
    fprintf (File, "///\n");
    fprintf (File, "/// Technology Library = %s\n",
             GnlHLibSourceFileName (GnlLib));
    fprintf (File,
              "\n");
    fprintf (File, "/// Top Level Module = %s\n",
             GnlName (GnlNetworkTopGnl (Nw)));
    fprintf (File,
              "\n");
    fprintf (File,
              "///\n");
    fprintf (File,
              "/// ======\n\n");
}

/*
/* GnlPrintVerilogNwHeaderWithLib
*/
/*
/* Prints official network header in the verilog ouptut file with lib. */
/* info. */
*/
void GnlPrintVerilogNwHeaderWithLib (File, Nw, GnlLib)
    FILE          *File;
    GNL_NETWORK   Nw;
    GNL_LIB       GnlLib;
{

    fprintf (File,
              "\n");
    fprintf (File,
              "// This Verilog file has been automatically generated \n");
    fprintf (File,
              "// by %s %s\n", GNL_TOOL_NAME, GNL_MAPIT_VERSION);
    fprintf (File,
              "\n");
    fprintf (File,
              "// Copyright (c) 1994-99 by AVANT! Corp.\n");
    fprintf (File,
```

gnlprint.c

```
    " // \n");
    fprintf (File,
        " // Date:");
    GnlPrintDate (File);
    fprintf (File, "\n");
    fprintf (File, " //\n");
    fprintf (File, " // Technology Library = %s\n",
        GnlHLibSourceFileName (GnlLib));
    fprintf (File,
        " // \n");
    fprintf (File, " // Top Level Module = %s\n",
        GnlName (GnlNetworkTopGnl (Nw)));
    fprintf (File, " // Gates Area      = %.4f\n", GnlNetworkArea (Nw));
    fprintf (File,
        " // \n");
    fprintf (File,
        " // ======\n\n");
}

/*-----
/* GnlPrintVerilogNwHeader
*/
/*-----
/* Prints official network header in the verilog ouptut file.
*/
void GnlPrintVerilogNwHeader (File, Nw)
FILE          *File;
GNL_NETWORK  Nw;
{
    fprintf (File,
        " // ======\n\n");
    fprintf (File,
        " // This Verilog file has been automatically generated \n");
    fprintf (File,
        " // by %s %s\n", GNL_TOOL_NAME, GNL_MAPIT_VERSION);
    fprintf (File,
        " // \n");
    fprintf (File,
        " // Copyright (c) 1994-99 by AVANT! Corp.\n");
    fprintf (File,
        " // \n");
    fprintf (File,
        " // Date:");
    GnlPrintDate (File);
    fprintf (File, "\n");
    fprintf (File, " //\n");
    fprintf (File,
        " // \n");
    fprintf (File, " // Top Level Module = %s\n",
        GnlName (GnlNetworkTopGnl (Nw)));
    fprintf (File,
        " // \n");
    fprintf (File,
        " // ======\n\n");
}
```

gnlprint.c

```
/*-----*/
/* GnlPrintVerilogGnlHeader */
/*-----*/
/* Prints a module header in the verilog ouptut file. */
/*-----*/
void GnlPrintVerilogGnlHeader (FILE, GnlLib, Gnl)
    FILE      *File;
    GNL_LIB   GnlLib;
    GNL       Gnl;
{
    BLIST     Components;

    Components = GnlComponents (Gnl);

    fprintf (File,
        "/*-----\n");
    fprintf (File, "/* Module: [%s]\n", GnlName (Gnl));
    fprintf (File, "/* Gates Area      = %.4f\n", GnlArea (Gnl));
    fprintf (File, "/* Max Gates Depth = %d\n", GnlDepth (Gnl));
    fprintf (File, "/* Nb. Instances    = %d\n", BListSize (Components));
    fprintf (File, "/* Nb. Flip-Flops   = %d\n", GnlNbDffs (Gnl));
    fprintf (File, "/* Nb. Latches      = %d\n", GnlNbLatches (Gnl));
    fprintf (File, "/* Nb. TriStates    = %d\n", GnlNbTristates (Gnl));
    fprintf (File, "/* Max FanOut       = %d\n", GnlMaxFanout (Gnl));
    fprintf (File, "/*\n");
    fprintf (File,
        "/*-----\n");
}

/*-----*/
/* GnlMakeCorrectName */
/*-----*/
char *GnlMakeCorrectName (Name)
    char *Name;
{
    int          L;
    int          i;

    L = strlen (Name);
    for (i=0; i<L; i++)
    {
        switch (Name[i]) {
            case '-':
            case '[':
            case ']':
                Name[i] = '_';
                break;
            default:
                break;
        }
    }
    return (Name);
}
```

gnlprint.c

```
/*-----*/
/* GnlPrintVerilogInterface */ */
/*-----*/
GNL_STATUS GnlPrintVerilogInterface (File, Gnl)
    FILE          *File;
    GNL           Gnl;
{
    int           i;
    GNL_VAR      PortI;
    char          *NewName;

    NewName = GnlMakeCorrectName (GnlName (Gnl));
    fprintf (File, "module %s (", NewName);
    if (BListSize (GnlListPorts (Gnl)) == 0)
    {
        fprintf (File, ");\n");
        return (GNL_OK);
    }

    for (i=0; i<BListSize (GnlListPorts (Gnl))-1; i++)
    {
        PortI = (GNL_VAR)BListElt (GnlListPorts (Gnl), i);
        fprintf (File, "%s , ", GnlPrintVarName (PortI));
    }
    PortI = (GNL_VAR)BListElt (GnlListPorts (Gnl), i);
    fprintf (File, "%s);\n", GnlPrintVarName (PortI));

    for (i=0; i<BListSize (GnlListPorts (Gnl)); i++)
    {
        PortI = (GNL_VAR)BListElt (GnlListPorts (Gnl), i);
        if (GnlVarRangeUndefined (PortI))
            fprintf (File, "%s %s;\n", GnlDirName (PortI),
                     GnlPrintVarName (PortI));
        else
            fprintf (File, "%s [%d:%d] %s;\n", GnlDirName (PortI),
                     GnlVarMsb (PortI), GnlVarLsb (PortI),
                     GnlPrintVarName (PortI));
    }

    return (GNL_OK);
}

/*-----*/
/* GnlPrintCFormatInterface */ */
/*-----*/
GNL_STATUS GnlPrintCFormatInterface (File, GnlName, ListClocks,
                                     ListInputs, ListOutputs,
                                     ListAsyncLow, ListAsyncHigh)
    FILE          *File;
    char          *GnlName;
    BLIST         ListClocks;
    BLIST         ListInputs;
    BLIST         ListOutputs;
    BLIST         ListAsyncLow;
    BLIST         ListAsyncHigh;
{
```

```

int i;
GNL_VAR ClockI;
GNL_VAR InputI;
GNL_VAR OutputI;
GNL_VAR VarI;
char *RangeStr;
BLIST ListAsync;
int MaxId;
int j;
int FirstInput;
int FirstOutput;

MaxId = 0;

/* Extracting the original async signals of the netlist */
if (BListCreate (&ListAsync))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListAsyncLow); i++)
{
    VarI = (GNL_VAR)BListElt (ListAsyncLow, i);
    if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
        if (BListAddElt (ListAsync, (int*)VarI))
            return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (ListAsyncHigh); i++)
{
    VarI = (GNL_VAR)BListElt (ListAsyncHigh, i);
    if (GnlVarType (VarI) == GNL_VAR_ORIGINAL)
        if (BListAddElt (ListAsync, (int*)VarI))
            return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (ListInputs); i++)
{
    VarI = (GNL_VAR)BListElt (ListInputs, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListOutputs); i++)
{
    VarI = (GNL_VAR)BListElt (ListOutputs, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListClocks); i++)
{
    VarI = (GNL_VAR)BListElt (ListClocks, i);
    if (GnlVarId (VarI) > MaxId)
        MaxId = GnlVarId (VarI);
}
for (i=0; i<BListSize (ListAsync); i++)
{

```

gnlprint.c

```
VarI = (GNL_VAR)BListElt (ListAsync, i);
if (GnlVarId (VarI) > MaxId)
    MaxId = GnlVarId (VarI);
}

FirstInput = FirstOutput = 1;

for (j=0; j<= MaxId; j++)
{
    for (i=0; i<BListSize (ListInputs); i++)
    {
        VarI = (GNL_VAR)BListElt (ListInputs, i);
        if (GnlVarId (VarI) == j)
        {
            if (FirstInput)
                fprintf (File, ".inputs ");
            FirstInput = 0;
            fprintf (File, "%s ", GnlVarName (VarI));
        }
    }
    for (i=0; i<BListSize (ListOutputs); i++)
    {
        VarI = (GNL_VAR)BListElt (ListOutputs, i);
        if (GnlVarId (VarI) == j)
        {
            if (FirstOutput)
                fprintf (File, "\n.outputs ");
            FirstOutput = 0;
            fprintf (File, "%s ", GnlVarName (VarI));
        }
    }
}

return (GNL_OK);
}

/*-----
/* GnlPrintLocalSignals
/*-----*/
GNL_STATUS GnlPrintLocalSignals (File, Gnl)
FILE           *File;
GNL            Gnl;
{
    int             i;
    int             j;
    GNL_VAR  VarI;
    GNL_VAR  VarJ;
    char          *RangeStr;
    BLIST     HashTableNames;
    BLIST     BucketI;

HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
```

```

        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);

            /* we do not define internally expanded variables */
            if (GnlVarType (VarJ) != GNL_VAR_INTERNAL)
                continue;

            if (GnlVarIsVss (VarJ) || GnlVarIsVdd (VarJ))
                continue;

            if (GnlVarGetStrFromRange (VarJ, &RangeStr))
                return (GNL_MEMORY_FULL);

            if (RangeStr)
                fprintf (File, "wire %s %s ;\n", RangeStr,
                        GnlPrintVarName (VarJ));
            else
                fprintf (File, "wire %s ;\n", GnlPrintVarName (VarJ));
        }
        fprintf (File, "\n");

        return (GNL_OK);
    }

/*-----*/
/* GnlPrintAssocActualName */
/*-----*/
/* Prints out the description of the actual port. This one can be either */
/* a signal (GNL_VAR) or a concatenate of signals (GNL_NODE with op.   */
/* GNL_CONCAT). */
/*-----*/
void GnlPrintAssocActualName (File, ActualPort)
    FILE           *File;
    GNL_VAR        ActualPort;
{
    int             i;
    GNL_VAR        Vari;
    BLIST          Sons;
    GNL_NODE       ActualNode;

    if (!ActualPort)
    {
        fprintf (File, "");
        return;
    }

    if (GnlVarIsVar (ActualPort))
    {
        if (GnlVarIsVss (ActualPort))
        {
            fprintf (File, "1'b0");
            return;
        }
        if (GnlVarIsVdd (ActualPort))

```

gnlprint.c

```
{  
    fprintf (File, "1'b1");  
    return;  
}  
fprintf (File, "%s ", GnlPrintVarName (ActualPort));  
return;  
}  
  
/* this is then a GNL_NODE with a GNL_CONCAT operator. */  
ActualNode = (GNL_NODE)ActualPort;  
Sons = GnlNodeSons (ActualNode);  
fprintf (File, "{}");  
for (i=0; i<BListSize (Sons)-1; i++)  
{  
    VarI = (GNL_VAR)BListElt (Sons, i);  
    fprintf (File, "%s ,", GnlPrintVarName (VarI));  
}  
VarI = (GNL_VAR)BListElt (Sons, i);  
fprintf (File, "%s }", GnlPrintVarName (VarI));  
}  
  
/*-----*/  
/* GnlPrintUserComponent */  
/*-----*/  
void GnlPrintUserComponent (File, UserComponent)  
FILE *File;  
GNL_USER_COMPONENT UserComponent;  
{  
    int i;  
    BLIST Parameters;  
    BLIST Interface;  
    GNL_ASSOC AssocI;  
  
    if (GnlUserComponentInstName (UserComponent))  
    {  
        if (GnlUserComponentInstName (UserComponent) [0] != '\\')  
            fprintf (File, "%s \\%s ",  
                    GnlUserComponentName (UserComponent),  
                    GnlUserComponentInstName (UserComponent));  
        else  
            fprintf (File, "%s %s ",  
                    GnlUserComponentName (UserComponent),  
                    GnlUserComponentInstName (UserComponent));  
    }  
    else  
    {  
        fprintf (File, "%s ", GnlUserComponentName (UserComponent));  
    }  
  
#ifdef USE_PARAMETERS  
    Parameters = GnlUserComponentParameters (UserComponent);  
    if (Parameters && BListSize (Parameters))  
    {  
        fprintf (File, "#(");  
    }
```

gnlprint.c

```
for (i=0; i<BListSize (Parameters); i++)
{
    /* to implement */
}
fprintf (File, " ");
#endif

fprintf (File, "(");

Interface = GnlUserComponentInterface (UserComponent);
for (i=0; i<BListSize (Interface)-1; i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    if (GnlAssocFormalPort (AssocI))
    {
        /* if the formal parameter is a string of char.          */
        if (GnlUserComponentFormalType
            (UserComponent) == GNL_FORMAL_CHAR)
            fprintf (File, ".%s (", GnlAssocFormalPort (AssocI));
        else
            /* Otherwise it is a GNL_VAR object                  */
            fprintf (File, ".%s (",
                    GnlPrintVarName ((GNL_VAR)GnlAssocFormalPort (AssocI)));

        GnlPrintAssocActualName (File, GnlAssocActualPort (AssocI));
        fprintf (File, ", ");
    }
    else
    {
        /* With no formal port definition we cannot have an      actual
         */
        /* port corresponding to a GNL_NODE.                      */
        GnlPrintAssocActualName (File, (GnlAssocActualPort (AssocI)));
        fprintf (File, ", ");
    }
}

AssocI = (GNL_ASSOC)BListElt (Interface, i);
if (GnlAssocFormalPort (AssocI))
{
    /* if the formal parameter is a string of char.          */
    if (GnlUserComponentFormalType (UserComponent) == GNL_FORMAL_CHAR)
        fprintf (File, ".%s (", GnlAssocFormalPort (AssocI));
    else
        /* Otherwise it is a GNL_VAR object                  */
        fprintf (File, ".%s (",
                GnlPrintVarName ((GNL_VAR)GnlAssocFormalPort (AssocI)));
    GnlPrintAssocActualName (File, GnlAssocActualPort (AssocI));
    fprintf (File, ")");
}
else
{
    /* With no formal port definition we cannot have an actual */
    /* port corresponding to a GNL_NODE.                      */
    GnlPrintAssocActualName (File, GnlAssocActualPort (AssocI));
    fprintf (File, ")");
}
```

```

    fprintf (File, ",\n");

}

/*-----*/
/* GnlNameIsPresent */
/*-----*/
/* This procedure verifies if the name 'Name' is present in the */
/* expression corresponding to 'BoolOpr'. */
/* Returns 1 if yes and 0 otherwise. */
/*-----*/
int GnlNameIsPresent (Name, BoolOpr)
    char          *Name;
    LIBC_BOOL_OPR  BoolOpr;
{
    text_buffer      *VarName;

    if (!BoolOpr)
        return (0);

    switch (LibBoolOprType (BoolOpr)) {
        /* terminal case of a signal. */
        case ID_B:
            VarName = LibBoolOprIdName (BoolOpr);
            if (!strcmp (VarName, Name))
                return (1);
            return (0);
            break;

        case ZERO_B:
            return (0);
            break;

        case ONE_B:
            return (0);
            break;

        /* case of binary Boolean operators */
        case XOR_B:
        case OR_B:
        case AND_B:
            if (GnlNameIsPresent (Name, LibBoolOprLeftSon (BoolOpr)))
                return (1);
            if (GnlNameIsPresent (Name, LibBoolOprRightSon (BoolOpr)))
                return (1);
            return (0);

        case NOT_B:
            if (GnlNameIsPresent (Name, LibBoolOprRightSon (BoolOpr)))
                return (1);
            return (0);

        default:
            fprintf (stderr,
                    "Operator unsupported in 'GnlNameIsPresent'!");
    }
}

```

```

        return (0);
    }

}

/*-----*/
/* GnlPrintSequentialComponent */
/*-----*/
void GnlPrintSequentialComponent (File, SeqCompo)
FILE           *File;
GNL_SEQUENTIAL_COMPONENT SeqCompo;
{
    LIBC_CELL          Cell;
    LIBC_PIN           Pins;
    LIBC_NAME_LIST ListPinName;
    GNL_VAR            InputVar;
    GNL_VAR            OutputVar;
    GNL_VAR            OutputVarBar;
    GNL_VAR            ClockVar;
    GNL_VAR            ClearVar;
    GNL_VAR            PresetVar;
    LIBC_BOOL_OPR     Input;
    char               *Output;
    char               *OutputBar;
    LIBC_BOOL_OPR     Clock;
    LIBC_BOOL_OPR     Clear;
    LIBC_BOOL_OPR     Preset;
    char               *ActualName;
    LIBC_BOOL_OPR     PinFunction;
    GNL_FORM_OUTPUT   FormOutput;
    GNL_VAR            ActualVar;
    GNL_FORM_OUTPUT   FormOutputBar;
    char               *IQ;
    char               *IQN;

    Cell = GnlSeqCompoInfoCell (SeqCompo);

    FormOutput = GnlSequentialCompoFormOutput (SeqCompo);
    Input = GnlSeqCompoInfoInput (SeqCompo);
    InputVar = GnlSequentialCompoInput (SeqCompo);
    Output = GnlSeqCompoInfoOutput (SeqCompo);
    OutputVar = GnlSequentialCompoOutput (SeqCompo);
    OutputBar = GnlSeqCompoInfoOutputBar (SeqCompo);
    OutputVarBar = GnlSequentialCompoOutputBar (SeqCompo);
    Clock = GnlSeqCompoInfoClock (SeqCompo);
    ClockVar = GnlSequentialCompoClock (SeqCompo);
    Clear = GnlSeqCompoInfoReset (SeqCompo);
    ClearVar = GnlSequentialCompoReset (SeqCompo);
    Preset = GnlSeqCompoInfoSet (SeqCompo);
    PresetVar = GnlSequentialCompoSet (SeqCompo);

    IQ = LibFFLatchQName (LibCellFFLatch (Cell));
    IQN = LibFFLatchQNNName (LibCellFFLatch (Cell));

    if (GnlEnvUseVerilogPrimitives ())
    {

```

gnlprint.c

```
switch (GnlSequentialCompoOp (SeqCompo)) {
    case GNL_DFF:
        fprintf (File, " dff ()";
        break;
    case GNL_DFFX:
        fprintf (File, " dffx ()";
        break;
    case GNL_DFF0:
        fprintf (File, " dff0 ()";
        break;
    case GNL_DFF1:
        fprintf (File, " dff1 ()";
        break;
    case GNL_LATCH:
        fprintf (File, " latch ()";
        break;
    case GNL_LATCHX:
        fprintf (File, " latchx ()";
        break;
    case GNL_LATCH0:
        fprintf (File, " latch0 ()";
        break;
    case GNL_LATCH1:
        fprintf (File, " latch1 ()";
        break;
}

if (!OutputVar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (OutputVar));
if (!OutputVarBar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (OutputVarBar));

if (!InputVar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (InputVar));

if (!ClockVar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (ClockVar));
if (GnlSequentialCompoClockPol (SeqCompo))
    fprintf (File, "1'b1 , ");
else
    fprintf (File, "1'b0 , ");

if (!PresetVar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (PresetVar));
if (GnlSequentialCompoSetPol (SeqCompo))
    fprintf (File, "1'b1 , ");

```

```

else
    fprintf (File, "1'b0 , ");

if (!ClearVar)
    fprintf (File, "1'b0 , ");
else
    fprintf (File, " %s , ", GnlNameEffectiveVar (ClearVar));
if (GnlSequentialCompoResetPol (SeqCompo))
    fprintf (File, "1'b1 );");
else
    fprintf (File, "1'b0 );");

fprintf (File, "\n");
return;
}

 Pins = LibCellPins (Cell);

if (!GnlSequentialCompoInstName (SeqCompo))
{
    if (OutputVar)
    {
        if (GnlVarName (OutputVar) [0] != '\\')
            fprintf (File, "%s \\%s (", LibCellName (Cell),
                     GnlVarName (OutputVar));
        else
            fprintf (File, "%s %s (", LibCellName (Cell),
                     GnlVarName (OutputVar));
    }
    else
    {
        if (GnlVarName (OutputVar) [0] != '\\')
            fprintf (File, "%s \\%s (", LibCellName (Cell),
                     GnlVarName (OutputVarBar));
        else
            fprintf (File, "%s %s (", LibCellName (Cell),
                     GnlVarName (OutputVarBar));
    }
}
else
{
    fprintf (File, "%s \\%s (", LibCellName (Cell),
             GnlSequentialCompoInstName (SeqCompo));
}

for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    ListPinName = LibPinName (Pins); /* we know it is a simple name */

    ActualVar = NULL;
    if (GnlNameIsPresent (LibNameListName (ListPinName),
                          Input))
        ActualVar = InputVar;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),
                               Clock))
        ActualVar = ClockVar;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),

```

gnlprint.c

```
        Clear))
    ActualVar = ClearVar;
else if (GnlNameIsPresent (LibNameListName (ListPinName),
                           Preset))
    ActualVar = PresetVar;
else
{
    /* It is may be an output. */                                */
    PinFunction = LibPinFunction (Pins);
    if (PinFunction)
    {
        if (GnlNameIsPresent (IQ, PinFunction))
            ActualVar = OutputVar;
        else if (GnlNameIsPresent (IQN, PinFunction))
            ActualVar = OutputVarBar;
    }
}

ActualName = NULL;
if (ActualVar)
    ActualName = GnlNameEffectiveVar (ActualVar);

if (LibPinNext (Pins))
    fprintf (File, ".%s (%s ), ", LibNameListName (ListPinName),
             (ActualName == NULL ? "" : ActualName));
else
    fprintf (File, ".%s (%s )", LibNameListName (ListPinName),
             (ActualName == NULL ? "" : ActualName));
}
fprintf (File, ");\n");
}

/*-----*/
/* GnlPrintTriStateComponent */                                */
/*-----*/
void GnlPrintTriStateComponent (File, TriStateCompo)
FILE                      *File;
GNL_TRISTATE_COMPONENT     TriStateCompo;
{
    LIBC_CELL           Cell;
    LIBC_PIN            Pins;
    LIBC_NAME_LIST      ListPinName;
    LIBC_BOOL_OPR       Input;
    GNL_VAR             InputVar;
    LIBC_BOOL_OPR       Enable;
    GNL_VAR             EnableVar;
    char                *Output;
    GNL_VAR             OutputVar;
    char                *ActualName;
    GNL_VAR             ActualVar;

    Cell = GnlTriStateCompoInfoCell (TriStateCompo);

    Input = GnlTriStateCompoInfoInput (TriStateCompo);
    InputVar = GnlTriStateInput (TriStateCompo);
```

gnlprint.c

```
Enable = GnlTriStateCompoInfoEnable (TriStateCompo);
EnableVar = GnlTriStateSelect (TriStateCompo);

Output = GnlTriStateCompoInfoOutput (TriStateCompo);
OutputVar = GnlTriStateOutput (TriStateCompo);

if (GnlEnvUseVerilogPrimitives ())
{
    fprintf (File, " tristate ());

    fprintf (File, " %s , ", GnlNameEffectiveVar (OutputVar));
    fprintf (File, " %s , ", GnlNameEffectiveVar (InputVar));

    if (!EnableVar)
        fprintf (File, "1'b0 , ");
    else
        fprintf (File, " %s , ", GnlNameEffectiveVar (EnableVar));
    if (GnlTriStateSelectPol (TriStateCompo))
        fprintf (File, "1'b1 );");
    else
        fprintf (File, "1'b0 );");

    fprintf (File, "\n");
    return;
}

Pins = LibCellPins (Cell);

if (!GnlTriStateInstName (TriStateCompo))
{
    if (GnlVarName (OutputVar) [0] != '\\')
        fprintf (File, "    %s \\%s (", LibCellName (Cell),
                 GnlVarName (OutputVar));
    else
        fprintf (File, "    %s %s (", LibCellName (Cell),
                 GnlVarName (OutputVar));
}
else
{
    fprintf (File, "    %s \\%s (", LibCellName (Cell),
             GnlTriStateInstName (TriStateCompo));
}

for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    ListPinName = LibPinName (Pins); /* we know it is a simpel name */
    ActualVar = NULL;
    if (GnlNameIsPresent (LibNameListName (ListPinName),
                          Input))
        ActualVar = InputVar;
    else if (GnlNameIsPresent (LibNameListName (ListPinName),
                               Enable))
        ActualVar = EnableVar;
    else
        ActualVar = OutputVar;

    ActualName = NULL;
```

gnlprint.c

```
if (ActualVar)
    ActualName = GnlNameEffectiveVar (ActualVar);

if (LibPinNext (Pins))
    fprintf (File, ".%s (%s) , ", LibNameListName (ListPinName),
             (ActualName == NULL ? "" : ActualName));
else
    fprintf (File, ".%s (%s) ", LibNameListName (ListPinName),
             (ActualName == NULL ? "" : ActualName));
}
fprintf (File, ");\n");
}

/*-----
/* GnlPrintBufComponent
/*-----*/
void GnlPrintBufComponent (File, BufCompo)
FILE                      *File;
GNL_BUF_COMPONENT          BufCompo;
{
    LIBC_CELL              Cell;
    LIBC_PIN                Pins;
    LIBC_NAME_LIST           ListPinName;
    LIBC_BOOL_OPR            Input;
    GNL_VAR                 InputVar;
    char                     *Output;
    GNL_VAR                 OutputVar;
    char                     *ActualName;
    GNL_VAR                 ActualVar;

    Cell = GnlBufCompoInfoCell (BufCompo);

    Input = GnlBufCompoInfoInput (BufCompo);
    InputVar = GnlBufInput (BufCompo);

    Output = GnlBufCompoInfoOutput (BufCompo);
    OutputVar = GnlBufOutput (BufCompo);

    if (GnlEnvUseVerilogPrimitives ())
    {
        fprintf (File, " buf ());
        fprintf (File, " %s , ", GnlNameEffectiveVar (OutputVar));
        fprintf (File, " %s );", GnlNameEffectiveVar (InputVar));

        fprintf (File, "\n");
        return;
    }

    Pins = LibCellPins (Cell);

    if (!GnlBufInstName (BufCompo))
    {
        fprintf (File, "%s \\\%s (", LibCellName (Cell),
                 GnlVarName (InputVar));
    }
}
```

gnlprint.c

```
else
{
    fprintf (File, "%s \\\%s (", LibCellName (Cell),
             GnlBufInstName (BufCompo));
}

for (; Pins != NULL; Pins = LibPinNext (Pins))
{
    ListPinName = LibPinName (Pins); /* we know it is a simpel name */
    ActualVar = NULL;
    if (GnlNameIsPresent (LibNameListName (ListPinName),
                          Input))
        ActualVar = InputVar;
    else
        ActualVar = OutputVar;

    ActualName = NULL;
    if (ActualVar)
        ActualName = GnlNameEffectiveVar (ActualVar);

    if (LibPinNext (Pins))
        fprintf (File, ".%s (%s ), ", LibNameListName (ListPinName),
                 (ActualName == NULL ? "" : ActualName));
    else
        fprintf (File, ".%s (%s )", LibNameListName (ListPinName),
                 (ActualName == NULL ? "" : ActualName));
}
fprintf (File, ");\n");
}

/*-----
 * GnlPrintComponent
 * -----
 * Prints out component instantiation.
 * -----
void GnlPrintComponent (File, Component)
FILE                  *File;
GNL_COMPONENT  Component;
{

switch (GnlComponentType (Component)) {
    case GNL_SEQUENTIAL_COMPO:
        GnlPrintSequentialComponent (File,
                                      (GNL_SEQUENTIAL_COMPONENT) Component);
        break;

    case GNL_USER_COMPO:
        GnlPrintUserComponent (File,
                               (GNL_USER_COMPONENT) Component);
        break;

    case GNL_TRISTATE_COMPO:
        GnlPrintTriStateComponent (File,
                                   (GNL_TRISTATE_COMPONENT) Component);
        break;

    case GNL_BUF_COMPO:
```

gnlprint.c

```
GnlPrintBufComponent (File, (GNL_BUF_COMPONENT) Component);
break;

case GNL_MACRO_COMPO:
break;

default:
    GnlError (12 /* Unknown component */);
}
}

/*-----*/
/* GnlPrintGnlComponents */
/*-----*/
void GnlPrintGnlComponents (File, Gnl)
FILE          *File;
GNL           Gnl;
{
    int             i;
    GNL_COMPONENT  ComponentI;

    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElts (GnlComponents (Gnl), i);
        GnlPrintComponent (File, ComponentI);
    }
}

/*-----*/
/* GnlPrintVerilogPrimitiveNode */
/*-----*/
void GnlPrintVerilogPrimitiveNode (File, Gnl, Var, Node)
FILE          *File;
GNL           Gnl;
GNL_VAR       Var;
GNL_NODE      Node;
{
    int             i;
    GNL_NODE       SonI;
    GNL_VAR        NewVar;

    switch (GnlNodeOp (Node)) {
        case GNL_CONSTANTE:
            fprintf (File, "    assign %s = ", GnlNameEffectiveVar (Var));
            if (GnlNodeSons (Node))
                fprintf (File, "1'b1 ;");
            else
                fprintf (File, "1'b0 ;");
            return;

        case GNL_VARIABLE:
            NewVar = (GNL_VAR) GnlNodeSons (Node);
            fprintf (File, "    assign %s = %s ;", GnlNameEffectiveVar
(Var),
                    GnlNameEffectiveVar (NewVar));
    }
}
```

gnlprint.c

```
    return;

case GNL_NOT:
    fprintf (File, " not (%s , ", GnlNameEffectiveVar (Var));
    break;

case GNL_OR:
    fprintf (File, " or (%s , ", GnlNameEffectiveVar (Var));
    break;

case GNL_AND:
    fprintf (File, " and (%s , ", GnlNameEffectiveVar (Var));
    break;

default:
    fprintf (stderr, " ERROR: unknow Boolean Operator\n");
    exit (1);
}

for (i=0; i<BListSize (GnlNodeSons (Node))-1; i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

    switch (GnlNodeOp (SonI)) {
        case GNL_CONSTANTE:
            if (GnlNodeSons (SonI))
                fprintf (File, "1'b1 , ");
            else
                fprintf (File, "1'b0 , ");
            break;

        case GNL_VARIABLE:
            NewVar = (GNL_VAR)GnlNodeSons (SonI);
            fprintf (File, "%s , ", GnlNameEffectiveVar (NewVar));
            break;

        default:
            fprintf (stderr,
                    " ERROR: Forbiden Boolean Operator here\n");
            exit (1);
    }
}

SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
switch (GnlNodeOp (SonI)) {
    case GNL_CONSTANTE:
        if (GnlNodeSons (SonI))
            fprintf (File, "1'b1 );");
        else
            fprintf (File, "1'b0 );");
        break;

    case GNL_VARIABLE:
        NewVar = (GNL_VAR)GnlNodeSons (SonI);
        fprintf (File, "%s );", GnlNameEffectiveVar (NewVar));
        break;
```

gnlprint.c

```
default:
    fprintf (stderr,
              " ERROR: Forbiden Boolean Operator here\n");
    exit (1);
}

/*
 * GnlPrintMapVerilog
 */
/* Prints out the verilog output description of 'Gnl' in file 'File'.
 */
GNL_STATUS GnlPrintMapVerilog (File, Gnl, GnlLib, BestInv)
FILE          *File;
GNL           Gnl;
GNL_LIB       GnlLib;
LIB_CELL      BestInv;
{
    char          *GnlName;
    int            i;
    GNL_VAR       VarI;
    GNL_FUNCTION  FunctionI;
    int            InstanceId;
    int            IndexList;

    InstanceId = 0;
    IndexList = 0;

    GnlName = GnlName (Gnl);

#ifndef PLIB
    GnlPrintVerilogGnlHeader (File, GnlLib, Gnl);
#endif

    if (GnlPrintVerilogInterface (File, Gnl))
        return (GNL_MEMORY_FULL);

    if (GnlPrintLocalSignals (File, Gnl))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        if (!FunctionI)
        {
            fprintf (stderr, " ERROR: <%s> has no function !\n",
                    GnlVarName (VarI));
            continue;
        }

        if (GnlEnvUseVerilogPrimitives ())
        {
```

gnlprint.c

```
        GnlPrintVerilogPrimitiveNode (File, Gnl, Vari,
                                      GnlFunctionOnSet (FunctionI));
    }
    else
    {
        if (GnlPrintMappedGnlNode (File, Gnl, BestInv, &InstanceId, Vari,
                                   GnlFunctionOnSet (FunctionI),
                                   &IndexList))
            return (GNL_MEMORY_FULL);
    }
    fprintf (File, "\n");
}

fprintf (File, "\n");

GnlPrintGnlComponents (File, Gnl);

fprintf (File, "\nendmodule\n\n");

return (GNL_OK);
}

/*-----*/
/* GnlPrintMapCFormat
 *-----*/
/*-----*/
GNL_STATUS GnlPrintMapCFormat (File, Gnl, GnlLib, BestInv,
                               ListClocks, ListInputs,
                               ListOutputs, ListAsyncLow, ListAsyncHigh)
{
    FILE          *File;
    GNL           Gnl;
    GNL_LIB       GnlLib;
    LIB_CELL      BestInv;
    BLIST         ListClocks;
    BLIST         ListInputs;
    BLIST         ListOutputs;
    BLIST         ListAsyncLow;
    BLIST         ListAsyncHigh;
}

    char          *GnlName;
    int           i;
    GNL_VAR       Vari;
    GNL_FUNCTION  FunctionI;
    int           InstanceId;
    int           IndexList;

    InstanceId = 0;
    IndexList = 0;

    GnlName = GnlName (Gnl);

    fprintf (File, ".model %s.opt\n", GnlName);

    if (GnlPrintCFormatInterface (File, GnlName, ListClocks,
                                 ListInputs, ListOutputs,
                                 ListAsyncLow, ListAsyncHigh))
```

gnlprint.c

```
    return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        Vari = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (Vari);
        GnlPrintCFormatMappedGnlNode (File, Gnl, BestInv, &InstanceId, Vari,
                                      GnlFunctionOnSet (FunctionI),
                                      &IndexList);
        fprintf (File, "\n");
    }

    fprintf (File, "\n.end\n");

    return (GNL_OK);
}

/*
 * GnlPrintMapVerilogFile
 */
/*
 * the file 'File' is already open. This procedure closes it.
 * Prints out the verilog description of the netlist 'Gnl' in file
 * 'File'. For each gates information are added thru parameters during
 * the instantiation of the gate.
 */
GNL_STATUS GnlPrintMapVerilogFile (File, Gnl, GnlLib)
{
    FILE          *File;
    GNL           Gnl;
    GNL_LIB       GnlLib;
    int           i;
    GNL_VAR       Vari;
    GNL_STATUS    GnlStatus;
    GNL_FUNCTION  FunctionI;
    LIB_CELL      BestInverter;

    BestInverter = GetBestAreaLibInverter (GnlLib);

    if (GnlStatus = GnlPrintMapVerilog (File, Gnl, GnlLib, BestInverter))
        return (GnlStatus);

    return (GNL_OK);
}

/*
 * GnlPrintMapCFormatFile
 */
GNL_STATUS GnlPrintMapCFormatFile (File, Gnl, GnlLib)
{
    FILE          *File;
    GNL           Gnl;
    GNL_LIB       GnlLib;
```

gnlprint.c

```
int          i;
GNL_VAR      VarI;
GNL_STATUS   GnlStatus;
GNL_FUNCTION FunctionI;
BLIST        ListClocks;
BLIST        ListInputs;
BLIST        ListOutputs;
BLIST        ListAsynchHigh;
BLIST        ListAsynchLow;
LIB_CELL     BestInverter;

if ((GnlStatus = GnlExtractClocks (Gnl, &ListClocks)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractAsynchSignals (Gnl, &ListAsynchHigh,
                                         &ListAsynchLow)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractInputs (Gnl, ListClocks, ListAsynchLow,
                                   ListAsynchHigh, &ListInputs)))
    return (GnlStatus);

if ((GnlStatus = GnlExtractOutputs (Gnl, ListClocks, ListAsynchLow,
                                   ListAsynchHigh, &ListOutputs)))
    return (GnlStatus);

BestInverter = GetBestAreaLibInverter (GnlLib);

GnlPrintMapCFormat (File, Gnl, GnlLib, BestInverter, ListClocks,
                    ListInputs, ListOutputs, ListAsynchLow,
                    ListAsynchHigh);

}

/*-----
/* GnlPrintVerilogNetworkRec                               */
/*-----*/
GNL_STATUS GnlPrintVerilogNetworkRec (Nw, Gnl, GnlLib, OutFile)
{
    GNL_NETWORK      Nw;
    GNL             Gnl;
    GNL_LIB         GnlLib;
    FILE            *OutFile;
    BLIST           Components;
    int              i;
    GNL_COMPONENT   ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL             GnlCompoI;

    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));
}
```

gnlprint.c

```
Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (!GnlCompoI)
        continue;

    if (GnlPrintVerilogNetworkRec (Nw, GnlCompoI, GnlLib, OutFile))
        return (GNL_MEMORY_FULL);
}

#ifndef OUTPUT_CFORMAT
if (GnlPrintMapCFormatFile (OutFile, Gnl, GnlLib))
    return (GNL_MEMORY_FULL);
#else
if (GnlPrintMapVerilogFile (OutFile, Gnl, GnlLib))
    return (GNL_MEMORY_FULL);
#endif

return (GNL_OK);
}

/*-----
/* GnlVarFlattenNode
*/
/* This procedure modified GNL_NOE 'Node' so that its sons are either */
/* GNL_CONSTANTE or GNL_VARIABLE. When a son is a Node Tree we create */
/* a new functions pointing on this tree and adds it in the current Gnl.*/
*/
GNL_STATUS GnlVarFlattenNode (Gnl, Node, NewNode)
    GNL          Gnl;
    GNL_NODE     Node;
    GNL_NODE *NewNode;
{
    int           i;
    GNL_NODE     SonI;
    GNL_NODE     Son;
    GNL_NODE     NewSon;
    GNL_VAR      NewVar;
    GNL_FUNCTION NewFunction;
    GNL_VAR      Var;
    GNL_FUNCTION Function;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
    {
        *NewNode = Node;
        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
```

```

{
    *NewNode = Node;
    return (GNL_OK);
}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);

    if ((GnlNodeOp (SonI) == GNL_CONSTANTE) ||
        (GnlNodeOp (SonI) == GNL_VARIABLE))
        continue;

    /* Otherwise there are two levels of equations then we create */
    /* a new variable.                                              */
    if (GnlCreateUniqueVar (Gnl, "P", &NewVar))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeForVar (Gnl, NewVar, NewNode))
        return (GNL_MEMORY_FULL);

    if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);

    SetGnlVarFunction (NewVar, NewFunction);
    SetGnlFunctionOnSet (NewFunction, SonI);

    BListElt (GnlNodeSons (Node), i) = (int)*NewNode;

    if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
        return (GNL_MEMORY_FULL);
}

*NewNode = Node;

return (GNL_OK);
}

/*
 * GnlFlattenBooleanFunctionsInNwRec
 */
GNL_STATUS GnlFlattenBooleanFunctionsInNwRec (Nw, Gnl)
    GNL_NETWORK Nw;
    GNL Gnl;
{
    int i;
    GNL_VAR VarI;
    GNL_FUNCTION Function;
    BLIST Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL GnlCompoI;
    GNL_NODE NewNode;

    /* Already processed.                                         */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))

```

gnlprint.c

```
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    Function = GnlVarFunction (VarI);
    if (GnlVarFlattenNode (Gnl, GnlFunctionOnSet (Function),
                           &NewNode, 0))
        return (GNL_MEMORY_FULL);
    SetGnlFunctionOnSet (Function, NewNode);
}

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    if (!GnlUserComponentGnlDef (UserCompoI))
        continue;

    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (GnlFlattenBooleanFunctionsInNwRec (Nw, GnlCompoI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlFlattenBooleanFunctionsInNw
*/
-----*/
GNL_STATUS GnlFlattenBooleanFunctionsInNw (Nw, Gnl)
    GNL_NETWORK      Nw;
    GNL             Gnl;
{

/* Resetting the Tag for recursive traversal. */
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

return (GnlFlattenBooleanFunctionsInNwRec (Nw, Gnl));
}

/*-----
/* GnlPrintVerilogNetwork
*/
-----*/
GNL_STATUS GnlPrintVerilogNetwork (Nw, GnlLib, OutFileName)
    GNL_NETWORK      Nw;
    GNL_LIB          GnlLib;
    char            *OutFileName;
{
```

gnlprint.c

```
FILE          *OutFile;
GNL           TopGnl;

if ((OutFile = fopen (OutFileName, "w")) == NULL)
{
    return (GNL_CANNOT_OPEN_OUTFILE);
}

TopGnl = GnlNetworkTopGnl (Nw);

/* The user asked to print out the verilog by using Verilog          */
/* primitives.                                                       */
if (GnlEnvUseVerilogPrimitives ())
{
    /* We replace the LIBC cells by the associated Boolean function */
    if (GnlReplaceAllComponentsWithLinkLibCells (Nw, TopGnl, GnlLib))
        return (GNL_MEMORY_FULL);
    if (GnlFlattenBooleanFunctionsInNw (Nw, TopGnl))
        return (GNL_MEMORY_FULL);

    GnlPrintVerilogNwHeader (OutFile, Nw);
}
else
{
    GnlPrintVerilogNwHeaderWithLib (OutFile, Nw, GnlLib);
}

SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

if (GnlPrintVerilogNetworkRec (Nw, TopGnl, GnlLib, OutFile))
    return (GNL_MEMORY_FULL);

fclose (OutFile);
return (GNL_OK);
}

/*
*-----*
* GnlPrintNetworkInfo
*-----*/
GNL_STATUS GnlPrintNetworkInfo (File, Nw, GnlLib)
FILE          *File;
GNL_NETWORK   Nw;
LIBC_LIB      GnlLib;
{
    GNL           TopGnl;
    double        GateArea;
    double        NetArea;
    int           NbDffs;
    int           NbLatches;
    int           NbTristates;
    int           NbBuffers;
    double        NbEqGates;
    double        SizeBasicEqGate;
    GNL_LIB       HGnlLib;
```

gnlprint.c

```
HGnlLib = (GNL_LIB)LibHook (GnlLib);

SizeBasicEqGate = GnlHLibAreaBasicEqGate (HGnlLib);

TopGnl = GnlNetworkTopGnl (Nw);

GnlGetNetworkPredefinedComponents (Nw, &NbDffs, &NbLatches,
                                   &NbTristates, &NbBuffers);
GateArea = GnlGetNetworkGateArea (Nw);

NetArea = GnlNetArea (TopGnl);

if (SizeBasicEqGate != 0.0)
{
    NbEqGates = GateArea/SizeBasicEqGate;
}

fprintf (File, "\n");
fprintf (File, " =====\n");
fprintf (File, "          GLOBAL NETLIST REPORT\n");
fprintf (File, " =====\n");
fprintf (File, "      o TOP-LEVEL           = %s\n", GnlName (TopGnl));
fprintf (File, "      o TECH.LIBRARY        = %s\n",
        GnlHLibName (HGnlLib));
fprintf (File, "      o GATES AREA         = %.2f\n", GateArea);
if (NetArea != 0.0)
{
    fprintf (File, "      o NETS AREA          = %.2f\n", NetArea);
    fprintf (File, "      o TOTAL NETLIST AREA = %.2f\n",
            GateArea+NetArea);
}
if (SizeBasicEqGate)
    fprintf (File, "      o NB. EQUIVALENT GATES = %.2f\n", NbEqGates);
if (GnlNetworkPeriod (Nw))
{
    fprintf (File, "      o BEST CLOCK PERIOD   = %.2f ns\n",
            GnlNetworkPeriod (Nw));
}

fprintf (File, "      o NB. FLIP-FLOPS       = %d\n", NbDffs);
fprintf (File, "      o NB. LATCHES         = %d\n", NbLatches);
fprintf (File, "      o NB. TRISTATES       = %d\n", NbTristates);
fprintf (File, "      o NB. BUFFERS         = %d\n", NbBuffers);
fprintf (File,
        " =====\n\n");

return (GNL_OK);
}

/*-----*/
```

gnlread.c

```
/*-----*/
/*
 *      File:          gnlread.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"

/*
 *-----*
 * GLOBAL VARIABLES
 *-----*/
GNL   G_CurrentGnl;
BLIST G_ListOfGnls;
int   G_GnlIsBig;

/*
 *-----*
 * GnlRead
 *-----*/
int GnlRead (FileName, ListGnls)
    char           *FileName;
    BLIST         *ListGnls;
{
    FILE           *GnlFile;
    char           *CopyName;
    int            Status;
    int            i;
    GNL           GnlI;

    if ((GnlFile = freopen (FileName , "r", stdin)) == NULL)
    {
        fprintf (stderr, " ERROR: cannot find file '%s'\n",
                 FileName);
        exit (1);
    }

    G_GnlIsBig = 1;

    if ((Status = gnlparse ()))
    {
        fclose (GnlFile);
        return (Status);
    }

    fclose (GnlFile);

    *ListGnls = G_ListOfGnls;
```

gnlread.c

```
for (i=0; i<BListSize (G_ListOfGnls); i++)
{
    GnlI = (GNL)BListElt (G_ListOfGnls, i);

    if (GnlStrCopy (FileName, &CopyName))
        return (GNL_MEMORY_FULL);

    SetGnlSourceFileName (GnlI, CopyName);

    /* a classical Gnl has been read ... */  

    SetGnlType (GnlI, GNL_USER);
}

if (BListSize (*ListGnls) == 0)
{
    fprintf (stderr, " ERROR: File '%s' has no component definition\n",
             FileName);
    exit (1);
}

return (0);
}

/*-----*/
/* GnlReadSmall */
/*-----*/
int GnlReadSmall (FileName, ListGnls)
{
    char          *FileName;
    BLIST         *ListGnls;
{
    FILE          *GnlFile;
    char          *CopyName;
    int           Status;
    int           i;
    GNL           GnlI;

    if ((GnlFile = freopen (FileName , "r", stdin)) == NULL)
    {
        fprintf (stderr, "# ERROR: cannot find file '%s'\n",
                 FileName);
        return (1);
    }

    G_GnlIsBig = 0;

    if ((Status = gnlparse ()))
    {
        fclose (GnlFile);
        return (Status);
    }

    fclose (GnlFile);

    *ListGnls = G_ListOfGnls;
```

gnlread.c

```
for (i=0; i<BListSize (G_ListOfGnls); i++)
{
    GnlI = (GNL)BListElt (G_ListOfGnls, i);

    if (GnlStrCopy (FileName, &CopyName))
        return (GNL_MEMORY_FULL);

    SetGnlSourceFileName (GnlI, CopyName);

    /* a leaf cell Gnl has been read ... */           */
    SetGnlType (GnlI, GNL_CELL);
}

return (0);
}

/*-----*/
```

gnlreport.c

```
/*-----*/
/*          File:      gnlreport.c
/*          Version:    1.1
/*          Modifications: -
/*          Documentation: -
/*          Description: *
/*-----*/
#include <stdio.h>

#ifndef MEMORY           /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "bbdd.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnllibc.h"
#include "gnlmap.h"
#include "gnloption.h"
#include "gnlreport.h"

#include "blist.e"

/*-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV   G_GnlEnv;

/*-----*/
/* DEFINE */
/*-----*/
#define HASH_TABLE_LIBC_CELLS_SIZE 1000
#define HASH_TABLE_MODULES_SIZE     1000

/*-----*/
/* GLOBAL */
/*-----*/
static BLIST   G_HashTableLibCells;
static BLIST   G_HashTableModules;

/*-----*/
/* GnlCreateDataReport */
/*-----*/
GNL_STATUS GnlCreateDataReport (DataReport)
    GNL_DATA_REPORT      *DataReport;
{

    if ((*DataReport = (GNL_DATA_REPORT)
        calloc (1, sizeof (GNL_DATA_REPORT_REC))) == NULL)
```

gnlreport.c

```
    return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
 * GnlAddOrGetLibcellInHashTable
 */
/*
 * we create a Data report information structure which will be attached *
 * to all the different cells used in the processed Gnl.          */
/*
GNL_STATUS GnlAddOrGetLibcellInHashTable (Cell, DataReport)
    LIBC_CELL           Cell;
    GNL_DATA_REPORT     *DataReport;
{
    char                 *CellName;
    int                  Key;
    BLIST                Bucket;
    int                  i;
    GNL_DATA_REPORT      CellreportI;
    BLIST                NewList;

    CellName = LibCellName (Cell);

    Key = KeyOfName (CellName, BListSize (G_HashTableLibCells));

    Bucket = (BLIST)BListElt (G_HashTableLibCells, Key);
    if (!Bucket)
    {
        if (BListCreateWithSize (1, &Bucket))
            return (GNL_MEMORY_FULL);

        BListElt (G_HashTableLibCells, Key) = (int)Bucket;
    }

    for (i=0; i<BListSize (Bucket); i++)
    {
        CellreportI = (GNL_DATA_REPORT)BListElt (Bucket, i);

        if (GnlDataReportCell (CellreportI) == Cell)
        {
            *DataReport = CellreportI;
            return (GNL_OK);
        }
    }

    if (GnlCreateDataReport (DataReport))
        return (GNL_MEMORY_FULL);

    SetGnlDataReportCell (*DataReport, Cell);
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    SetGnlDataReportMaxFanoutCompo (*DataReport, NewList);

    if (BListAddElt (Bucket, (int)(*DataReport)))

```

gnlreport.c

```
    return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlAddOrGetModuleInHashTable
*/
-----*/
GNL_STATUS GnlAddOrGetModuleInHashTable (Gnl, DataReport)
    GNL                      Gnl;
    GNL_DATA_REPORT           *DataReport;
{
    char                      *CellName;
    int                       Key;
    BLIST                     Bucket;
    int                       i;
    GNL_DATA_REPORT           CellreportI;

    CellName = GnlName (Gnl);

    Key = KeyOfName (CellName, BListSize (G_HashTableModules));

    Bucket = (BLIST)BListElt (G_HashTableModules, Key);
    if (!Bucket)
    {
        if (BListCreateWithSize (1, &Bucket))
            return (GNL_MEMORY_FULL);

        BListElt (G_HashTableModules, Key) = (int)Bucket;
    }

    for (i=0; i<BListSize (Bucket); i++)
    {
        CellreportI = (GNL_DATA_REPORT)BListElt (Bucket, i);

        if (GnlDataReportGnl (CellreportI) == Gnl)
        {
            *DataReport = CellreportI;
            return (GNL_OK);
        }
    }

    if (GnlCreateDataReport (DataReport))
        return (GNL_MEMORY_FULL);

    SetGnlDataReportGnl (*DataReport, Gnl);

    if (BListAddElt (Bucket, (int)(*DataReport)))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlGetFanoutFromSeqCompo
*/
-----*/
```

gnlreport.c

```
int GnlGetFanoutFromSeqCompo (SeqCompo)
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
{
    int FanOut;
    GNL_VAR OutVar;
    GNL_VAR OutVarBar;

    OutVar = GnlSequentialCompoOutput (SeqCompo);
    OutVarBar = GnlSequentialCompoOutputBar (SeqCompo);

    FanOut = 0;

    if (OutVar && (FanOut < (int)GnlVarHook (OutVar)))
        FanOut = (int)GnlVarHook (OutVar);

    if (OutVarBar && (FanOut < (int)GnlVarHook (OutVarBar)))
        FanOut = (int)GnlVarHook (OutVarBar);

    return (FanOut);
}

/*-----
/* GnlGetFanoutFromTristateCompo
*/
-----*/
int GnlGetFanoutFromTristateCompo (TristateCompo)
    GNL_TRISTATE_COMPONENT TristateCompo;
{
    int FanOut;
    GNL_VAR OutVar;

    OutVar = GnlTriStateOutput (TristateCompo);

    FanOut = 0;

    if (OutVar && (FanOut < (int)GnlVarHook (OutVar)))
        FanOut = (int)GnlVarHook (OutVar);

    return (FanOut);
}

/*-----
/* GnlGetFanoutFromBufCompo
*/
-----*/
int GnlGetFanoutFromBufCompo (BufCompo)
    GNL_BUF_COMPONENT BufCompo;
{
    int FanOut;
    GNL_VAR OutVar;

    OutVar = GnlBufOutput (BufCompo);

    FanOut = 0;

    if (OutVar && (FanOut < (int)GnlVarHook (OutVar)))
```

gnlreport.c

```
FanOut = (int)GnlVarHook (OutVar);

    return (FanOut);
}

/*-----
/* GnlGetPinCellWithName
/*-----*/
LIBC_PIN GnlGetPinCellWithName (Cell, Name)
    LIBC_CELL          Cell;
    char              *Name;
{
    LIBC_PIN      Pins;
    LIBC_NAME_LIST ListPinName;

    Pins = LibCellPins (Cell);
    for (;Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simple name */
        if (!strcmp (LibNameListName (ListPinName), Name))
            return (Pins);
    }

    return (NULL);
}

/*-----
/* GnlGetFanoutFromUserCompo
/*-----*/
int GnlGetFanoutFromUserCompo (UserCompo)
    GNL_USER_COMPONENT      UserCompo;
{
    int             i;
    int             FanOut;
    char            *FormalI;
    GNL_VAR        ActualI;
    BLIST          Interface;
    GNL_VAR        VarJ;
    BLIST          ListSplitActuals;
    int             j;
    GNL_ASSOC      AssocI;
    LIBC_CELL      Cell;
    LIBC_PIN       PinFormal;

    Cell = GnlUserComponentCellDef (UserCompo);
    Interface = GnlUserComponentInterface (UserCompo);
    FanOut = 0;

    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        FormalI = GnlAssocFormalPort (AssocI);
        ActualI = GnlAssocActualPort (AssocI);
        if (!ActualI)
            continue;
```

```

PinFormal = GnlGetPinCellWithName (Cell, FormalI);

if (GnlVarIsVar (ActualI))
{
    if ((LibPinDirection (PinFormal) == OUTPUT_E) ||
        (LibPinDirection (PinFormal) == INOUT_E))
    {
        if (ActualI && (FanOut < (int)GnlVarHook (ActualI)))
            FanOut = (int)GnlVarHook (ActualI);
    }
}
else
{
    ListSplitActuals = GnlNodeSons ((GNL_NODE)ActualI);
    for (j=0; j<BListSize (ListSplitActuals); j++)
    {
        VarJ = (GNL_VAR)BListElt (ListSplitActuals, j);
        if ((LibPinDirection (PinFormal) == OUTPUT_E) ||
            (LibPinDirection (PinFormal) == INOUT_E))
        {
            if (VarJ && (FanOut < (int)GnlVarHook (VarJ)))
                FanOut = (int)GnlVarHook (VarJ);
        }
    }
}

return (FanOut);
}

/*-----
/* GnlGetLibCells
/*-----*/
GNL_STATUS GnlGetLibCells (Gnl)
    GNL          Gnl;
{
    int                  i;
    GNL_COMPONENT       ComponentI;
    GNL_USER_COMPONENT UserCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TristateCompo;
    GNL_BUF_COMPONENT BufCompo;
    LIBC_CELL           Cell;
    GNL                 SubGnl;
    GNL_DATA_REPORT     DataReport;
    double              CellArea;
    int                 FanOut;

    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);

        switch (GnlComponentType (ComponentI)) {
            case GNL_USER_COMPO:
                UserCompo = (GNL_USER_COMPONENT)ComponentI;

```

```

/* If not a black box or cell we do take it into */
/* account. */
Cell = GnlUserComponentCellDef (UserCompo);
if (!Cell)
{
    SubGnl = GnlUserComponentGnlDef (UserCompo);
    if (SubGnl)
    {
        if (GnlGetLibCells (SubGnl))
            return (GNL_MEMORY_FULL);
    }
}
else
{
    FanOut = GnlGetFanoutFromUserCompo (UserCompo);
}
break;

case GNL_SEQUENTIAL_COMPO:
    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    FanOut = GnlGetFanoutFromSeqCompo (SeqCompo);
    break;

case GNL_TRISTATE_COMPO:
    TristateCompo = (GNL_TRISTATE_COMPONENT) ComponentI;
    Cell = GnlTriStateCompoInfoCell (TristateCompo);
    FanOut = GnlGetFanoutFromTriStateCompo (TristateCompo);
    break;

case GNL_BUF_COMPO:
    BufCompo = (GNL_BUF_COMPONENT) ComponentI;
    Cell = GnlBufCompoInfoCell (BufCompo);
    FanOut = GnlGetFanoutFromBufCompo (BufCompo);
    break;

}

if (!Cell)
    continue;

/* We create or get the Data report information attached to the */
/* cell 'Cell'. */
if (GnlAddOrGetLibcellInHashTable (Cell, &DataReport))
    return (GNL_MEMORY_FULL);

SetGnlDataReportNb (DataReport, GnlDataReportNb (DataReport)+1);
CellArea = (double) LibCellArea (Cell);
SetGnlDataReportArea (DataReport,
                      GnlDataReportArea (DataReport)+CellArea);
/* we store the cells with max fanout */
if (!BListSize (GnlDataReportMaxFanoutCompo (DataReport)))
{
    if (BListAddElt (GnlDataReportMaxFanoutCompo (DataReport),
                     (int) ComponentI))
        return (GNL_MEMORY_FULL);
    continue;
}

```

```

    }

/* If the current Cell 'cell' has a higher Fanout than the max */
/* current Fanout we store it and rebuild the list of max fanout*/
/* cells. */
if (FanOut > GnlDataReportMaxFanout (DataReport))
{
    SetGnlDataReportMaxFanout (DataReport, FanOut);
    BSize (GnlDataReportMaxFanoutCompo (DataReport)) = 0;
    if (BListAddElt (GnlDataReportMaxFanoutCompo (DataReport),
                      (int)ComponentI))
        return (GNL_MEMORY_FULL);
}
else if (FanOut == GnlDataReportMaxFanout (DataReport))
{
    if (BListAddElt (GnlDataReportMaxFanoutCompo (DataReport),
                      (int)ComponentI))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlGetModules
 */
GNL_STATUS GnlGetModules (Gnl)
{
    GNL          Gnl;
    int          i;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TristateCompo;
    LIBC_CELL      Cell;
    GNL           SubGnl;
    GNL_DATA_REPORT DataReport;

    if (GnlAddOrGetModuleInHashTable (Gnl, &DataReport))
        return (GNL_MEMORY_FULL);

    SetGnlDataReportNb (DataReport, GnlDataReportNb (DataReport)+1);
    SetGnlDataReportArea (DataReport,
                          GnlDataReportArea (DataReport)+GnlArea (Gnl));

    for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);

        switch (GnlComponentType (ComponentI)) {
            case GNL_USER_COMPO:
                UserCompo = (GNL_USER_COMPONENT)ComponentI;
                Cell = GnlUserComponentCellDef (UserCompo);
                if (!Cell)

```

```

    {
        SubGnl = GnlUserComponentGnlDef (UserCompo);
        if (SubGnl)
        {
            if (GnlGetModules (SubGnl))
                return (GNL_MEMORY_FULL);
        }
    }
    break;

    default:
        break;
}
}

return (GNL_OK);
}

/*-----
/* GnlReportCells
*/
-----*/
GNL_STATUS GnlReportCells (Nw)
    GNL_NETWORK Nw;
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;

    if (BListCreateWithSize (HASH_TABLE_LIBC_CELLS_SIZE,
                           &G_HashTableLibCells))
        return (GNL_MEMORY_FULL);
    BSize (G_HashTableLibCells) = HASH_TABLE_LIBC_CELLS_SIZE;

    TopGnl = GnlNetworkTopGnl (Nw);

    if ((GnlStatus = GnlGetLibCells (TopGnl)))
        return (GnlStatus);

    return (GNL_OK);
}

/*-----
/* GnlReportModules
*/
-----*/
GNL_STATUS GnlReportModules (Nw)
    GNL_NETWORK Nw;
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;

    if (BListCreateWithSize (HASH_TABLE_MODULES_SIZE,
                           &G_HashTableModules))
        return (GNL_MEMORY_FULL);
    BSize (G_HashTableModules) = HASH_TABLE_MODULES_SIZE;
}

```

gnlreport.c

```
TopGnl = GnlNetworkTopGnl (Nw);

if ((GnlStatus = GnlGetModules (TopGnl)))
    return (GnlStatus);

return (GNL_OK);
}

/*-----
/* GnlPrintReportCells
/*-----*/
void GnlPrintReportCells (Nw)
    GNL_NETWORK Nw;
{
    GNL                      TopGnl;
    int                       i;
    int                       j;
    BLIST                     BucketI;
    double                    TotalArea;
    double                    TotalDesignArea;
    GNL_DATA_REPORT           DataReportJ;
    LIBC_CELL                 Cell;
    char                      *CellName;
    int                       k;
    GNL_COMPONENT              ComponentK;
    GNL_USER_COMPONENT          UserCompo;
    GNL_TRISTATE_COMPONENT      TriCompo;
    GNL_BUF_COMPONENT            BufCompo;
    GNL_SEQUENTIAL_COMPONENT     SeqCompo;
    char                      *InstName;
    BLIST                     ListFanoutCompo;
    int                       NbInstances;

TopGnl = GnlNetworkTopGnl (Nw);

/* Computing first the total design gate area */
TotalDesignArea = 0.0;
for (i=0; i<BListSize (G_HashTableLibCells); i++)
{
    BucketI = (BLIST)BListElt (G_HashTableLibCells, i);
    if (!BucketI)
        continue;

    for (j=0; j<BListSize (BucketI); j++)
    {
        DataReportJ = (GNL_DATA_REPORT)BListElt (BucketI, j);
        TotalDesignArea += GnlDataReportArea (DataReportJ);
    }
}

fprintf (stderr, "\n");
fprintf (stderr, " REPORT CELLS : DESIGN = [%s]\n",
        GnlName (TopGnl));
fprintf (stderr, " -----");
for (i=0; i<strlen (GnlName (TopGnl)); i++)
```

gnlreport.c

```
    fprintf (stderr, "-");
    fprintf (stderr, "-\n");
    fprintf (stderr, "\n");
    fprintf (stderr,
"  Cell           Number      Cell_Area     Max_FanOut      Total_Area   %c
Design\n", '%');
    fprintf (stderr,
"  -----\n");
    \n");

    TotalArea = 0.0;
    NbInstances = 0;

    for (i=0; i<BListSize (G_HashTableLibCells); i++)
    {
        BucketI = (BLIST)BListElt (G_HashTableLibCells, i);
        if (!BucketI)
            continue;

        for (j=0; j<BListSize (BucketI); j++)
        {
            DataReportJ = (GNL_DATA_REPORT)BListElt (BucketI, j);
            Cell = GnlDataReportCell (DataReportJ);
            CellName = LibCellName (Cell);
            fprintf (stderr, "  ");
            GnlPrintFormatSignalName (stderr, CellName, 12);

            TotalArea += GnlDataReportArea (DataReportJ);

            /* Dirty fix to not print 0 for max fanout. To change ! */
            if (!GnlDataReportMaxFanout (DataReportJ))
                GnlDataReportMaxFanout (DataReportJ) = 1;

            fprintf (stderr,
                    "  %6d    %11.2f      %5d  %13.2f    %4.1f %c\n",
                    GnlDataReportNb (DataReportJ),
                    GnlDataReportArea (DataReportJ),
                    GnlDataReportMaxFanout (DataReportJ), TotalArea,
                    (100*GnlDataReportArea (DataReportJ))/TotalDesignArea, '%');

            NbInstances += GnlDataReportNb (DataReportJ);
        }
    }

    fprintf (stderr,
"  -----\n");
    \n");
    fprintf (stderr,
            "  Nb.Cells      %6d                                %13.2f\n",
            NbInstances, TotalDesignArea);
    fprintf (stderr, "\n");

    if (!GnlEnvPrintMaxFanoutVars ())
        return;
```

gnlreport.c

```
fprintf (stderr, " Cell Name      Instance Name      Max FanOut\n");
fprintf (stderr,
" -----
\n");
for (i=0; i<BListSize (G_HashTableLibCells); i++)
{
    BucketI = (BLIST)BListElt (G_HashTableLibCells, i);
    if (!BucketI)
        continue;

    for (j=0; j<BListSize (BucketI); j++)
    {
        DataReportJ = (GNL_DATA_REPORT)BListElt (BucketI, j);
        ListFanoutCompo = GnlDataReportMaxFanoutCompo (DataReportJ);
        Cell = GnlDataReportCell (DataReportJ);
        CellName = LibCellName (Cell);
        fprintf (stderr, " ");
        GnlPrintFormatSignalName (stderr, CellName, 12);
        fprintf (stderr, "\n");

        for (k=0; k<BListSize (ListFanoutCompo); k++)
        {
            ComponentK = (GNL_COMPONENT)BListElt (ListFanoutCompo, k);
            switch (GnlComponentType (ComponentK)) {
                case GNL_USER_COMPO:
                    UserCompo = (GNL_USER_COMPONENT)ComponentK;
                    InstName = GnlUserComponentInstName (UserCompo);
                    break;

                case GNL_SEQUENTIAL_COMPO:
                    SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentK;
                    InstName = GnlSequentialCompoInstName
(SeqCompo);
                    break;

                case GNL_TRISTATE_COMPO:
                    TriCompo = (GNL_TRISTATE_COMPONENT)ComponentK;
                    InstName = GnlTriStateInstName (TriCompo);
                    break;

                case GNL_BUF_COMPO:
                    BufCompo = (GNL_BUF_COMPONENT)ComponentK;
                    InstName = GnlBufInstName (BufCompo);
                    break;

            }
            fprintf (stderr, "           ");
            GnlPrintFormatSignalName (stderr, InstName, 12);
            fprintf (stderr, " %d\n",
                    GnlDataReportMaxFanout (DataReportJ));
        }
    }
}
fprintf (stderr,
" -----
\n");
fprintf (stderr, "\n");
```

gnlreport.c

```
}

/*-----*/
/* GnlPrintReportModules */
/*-----*/
GNL_STATUS GnlPrintReportModules (Nw)
    GNL_NETWORK Nw;
{
    GNL          TopGnl;
    int          i;
    int          j;
    BLIST        BucketI;
    double       TotalArea;
    double       TotalDesignArea;
    GNL_DATA_REPORT DataReportJ;
    char         *CellName;
    int          k;
    GNL          Gnl;

    TopGnl = GnlNetworkTopGnl (Nw);

    /* Computing first the total design gate area */
    TotalDesignArea = 0.0;
    for (i=0; i<BListSize (G_HashTableModules); i++)
    {
        BucketI = (BLIST)BListElt (G_HashTableModules, i);
        if (!BucketI)
            continue;

        for (j=0; j<BListSize (BucketI); j++)
        {
            DataReportJ = (GNL_DATA_REPORT)BListElt (BucketI, j);
            TotalDesignArea += GnlDataReportArea (DataReportJ);
        }
    }

    fprintf (stderr, "\n");
    fprintf (stderr, " REPORT MODULES : DESIGN = [%s]\n",
             GnlName (TopGnl));
    fprintf (stderr, " -----");
    for (i=0; i<strlen (GnlName (TopGnl)); i++)
        fprintf (stderr, "-");
    fprintf (stderr, "-\n");
    fprintf (stderr, "\n");
    fprintf (stderr,
" Module           Number     Cell_Area   Fanout   Wire_Path   Total_Area   %c
Design\n", '%');
    fprintf (stderr,
" -----");
    -\n");
}

TotalArea = 0.0;
```

gnlreport.c

```
for (i=0; i<BListSize (G_HashTableModules); i++)
{
    BucketI = (BLIST)BListElt (G_HashTableModules, i);
    if (!BucketI)
        continue;

    for (j=0; j<BListSize (BucketI); j++)
    {
        DataReportJ = (GNL_DATA_REPORT)BListElt (BucketI, j);
        Gnl = GnlDataReportGnl (DataReportJ);
        CellName = GnlName (Gnl);
        fprintf (stderr, " ");
        if (strlen (CellName) >= 14)
        {
            fprintf (stderr, "..");
            for (k=strlen (CellName)-12; k<strlen (CellName); k++)
                fprintf (stderr, "%c", CellName[k]);
        }
        else
        {
            for (k=0; k<strlen (CellName); k++)
                fprintf (stderr, "%c", CellName[k]);
            while (k<14)
            {
                fprintf (stderr, " ");
                k++;
            }
        }
    }

    TotalArea += GnlDataReportArea (DataReportJ);

    fprintf (stderr,
             " %6d %11.2f %4d %4d %13.2f %4.1f %c\n",
             GnlDataReportNb (DataReportJ),
             GnlDataReportArea (DataReportJ),
             GnlMaxFanout (Gnl), GnlDepth (Gnl), TotalArea,
             (100*GnlDataReportArea (DataReportJ))/TotalDesignArea, '%');
}

}

fprintf (stderr,
" -----\n");
fprintf (stderr,
"%13.2f \n", TotalDesignArea);

}

/*
 * GnlReportDataAfterSynthesis
 */
GNL_STATUS GnlReportDataAfterSynthesis (Nw, GnlLib)
{
    GNL_NETWORK Nw;
    LIBC_LIB GnlLib;
{
    GNL_STATUS      GnlStatus;
```

gnlreport.c

```
GNL          TopGnl;

switch (GnlEnvReportData()) {
    case GNL_REPORT_MODULES:
        if (GnlReportModules (Nw))
            return (GNL_MEMORY_FULL);

        GnlPrintReportModules (Nw);
        break;

    case GNL_REPORT_CELLS:
        GnlComputeMaxFanoutInNetwork (Nw);

        if (GnlReportCells (Nw))
            return (GNL_MEMORY_FULL);

        GnlPrintReportCells (Nw);
        break;

    case GNL_REPORT_TIMINGS:
        if ((GnlStatus = GnlTimingEstimateAfterSynthesis (Nw,
                                                       GnlLib)))
            return (GnlStatus);
        break;

    case GNL_REPORT_POWERS:
        if ((GnlStatus = GnlPowerEstimateAfterSynthesis (Nw,
                                                       GnlLib)))
            return (GnlStatus);
        break;

    default:
        break;
}

return (GNL_OK);
}

/*
 * GnlReportDataForEstimation
 */
GNL_STATUS GnlReportDataForEstimation (Nw, GnlLib)
    GNL_NETWORK    Nw;
    LIBC_LIB       GnlLib;
{
    GNL_STATUS     GnlStatus;
    GNL           TopGnl;
    GNL_LIB        HGnlLib;

    fprintf (stderr, "\n");
    fprintf (stderr, " Performing estimation..\n");

    switch (GnlEnvReportData()) {
        case GNL_REPORT_MODULES:
            GnlGetGnlMapAreaInNw (Nw);
```

gnlreport.c

```
if (GnlReportModules (Nw))
    return (GNL_MEMORY_FULL);

GnlPrintReportModules (Nw);
break;

case GNL_REPORT_CELLS:
    GnlComputeMaxFanoutInNetwork (Nw);

    if (GnlReportCells (Nw))
        return (GNL_MEMORY_FULL);

    GnlPrintReportCells (Nw);
    break;

case GNL_REPORT_TIMINGS:
    if ((GnlStatus = GnlTimingEstimate (Nw, GnlLib)))
        return (GnlStatus);
    break;

case GNL_REPORT POWERS:
    if ((GnlStatus = GnlPowerEstimate (Nw, GnlLib)))
        return (GnlStatus);
    break;

default:
    break;
}

fprintf (stderr, "\n");
fprintf (stderr, " Estimation Done !\n");
fprintf (stderr, "\n");

return (GNL_OK);
}

/* ----- EOF ----- */
```

gnlreport.h

```
/*-----*/
/*
 *      File:          gnlreport.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*-----*/
/*-----*/
/* GNL_DATA_REPORT
 */
/*-----*/
typedef struct GNL_DATA_REPORT_STRUCT {
    void             *Cell;
    int              Number;
    double           Area;
    int              MaxFanout;
    BLIST            MaxFanoutCompo;
}
    GNL_DATA_REPORT_REC, *GNL_DATA_REPORT;

#define GnlDataReportCell(n)          ((LIBC_CELL)((n)->Cell))
#define GnlDataReportGnl(n)          ((GNL)((n)->Cell))
#define GnlDataReportNb(n)           ((n)->Number)
#define GnlDataReportArea(n)          ((n)->Area)
#define GnlDataReportMaxFanout(n)    ((n)->MaxFanout)
#define GnlDataReportMaxFanoutCompo(n) ((n)->MaxFanoutCompo)

#define SetGnlDataReportCell(n,c)     ((n)->Cell = c);
#define SetGnlDataReportGnl(n,c)      ((n)->Cell = c);
#define SetGnlDataReportNb(n,c)       ((n)->Number = c);
#define SetGnlDataReportArea(n,c)     ((n)->Area = c);
#define SetGnlDataReportMaxFanout(n,c) ((n)->MaxFanout = c);
#define SetGnlDataReportMaxFanoutCompo(n,c) ((n)->MaxFanoutCompo = c);

/* ----- EOF ----- */
```

gnlstr.c

```
/*-----*/
/*
 *      File:          gnlstr.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"

/* -----
/* GnlStrCopy                               */
/* -----*/
/* Create a physical duplication of the string 'str' and returns GNL_OK */
/* if everything is ok and GNL_MEMORY_FULL is the memory allocation was */
/* not feasible.                           */
/* -----*/
GNL_STATUS GnlStrCopy (str, NewStr)
    char *str;
    char **NewStr;
{

    if ((*NewStr = (char*)GNL_CALLOC ((unsigned) 1,
                                      (unsigned) (strlen (str) + 1))) == NULL)
        return (GNL_MEMORY_FULL);

    (void) strcpy (*NewStr, str);

    return (GNL_OK);
}

/* -----
/* GnlStrAppendStrCopy                      */
/* -----*/
/* Create a new string stored in 'NewStr' which is the cat of 'str1' and */
/* 'str2'.                                */
/* -----*/
GNL_STATUS GnlStrAppendStrCopy (str1, str2, NewStr)
    char    *str1;
    char    *str2;
    char    **NewStr;
{

    if ((*NewStr = (char*)GNL_CALLOC ((unsigned) 1,
                                      (unsigned) (strlen (str1) + strlen (str2) + 1))) == NULL)
        return (GNL_MEMORY_FULL);

    sprintf (*NewStr, "%s%s", str1, str2);

    return (GNL_OK);
}
```

gnlstr.c

```
}

/*
 * -----
 * GnlStrAppendIntCopy
 * -----
 * Create a new string stored in 'NewStr' which is the cat of 'str1' and
 * integer 'Index'
 * -----
 * WARNING: index should not exceed 30 digits
 * -----
 */
GNL_STATUS GnlStrAppendIntCopy (str1, Index, NewStr)
    char *str1;
    int     Index;
    char **NewStr;
{
    char     *Str;

    if ((Str = (char*)GNL_CALLOC ((unsigned) 1, (unsigned) (strlen (str1) +
                                              30 + 1))) == NULL)
        return (GNL_MEMORY_FULL);

    sprintf (Str, "%s%d", str1, Index);

    if ((*NewStr = (char*)GNL_CALLOC ((unsigned) 1, (unsigned) (strlen (Str) +
1
                                              ))) == NULL)
        return (GNL_MEMORY_FULL);

    (void) strcpy (*NewStr, Str);

    free (Str);

    return (GNL_OK);
}

/*
 * -----
 * GnlStrReverse
 * -----
 */
GNL_STATUS GnlStrReverse (Str)
    char *Str;
{
    char     *Str1;
    char     *Ptr1;
    char     *Ptr;

    if ((Str1 = (char*)GNL_CALLOC ((unsigned) (strlen (Str) + 1),
                                    (unsigned) sizeof (char))) == NULL)
        return (GNL_MEMORY_FULL);

    (void) strcpy (Str1, Str);
    Ptr1 = Str1 + strlen (Str1) - 1;
    Ptr = Str;
    while (Ptr1 >= Str1)
    {
        (*Ptr++) = (*Ptr1--);
```

gnlstr.c

```
}

free (Str1);

return (GNL_OK);
}

/*
 * GnlStrDecimalToBin
 */
GNL_STATUS GnlDecimalToStrBin (Decim, Range, StrBin)
    int          Decim;
    int          Range;
    char        **StrBin;
{
    int          Exp = 0;
    int          BInf = 1;
    int          Aux = Decim;
    int          i;

    if ((*StrBin = (char*) GNL_CALLOC (Range+1, sizeof (char))) == NULL)
        return (GNL_MEMORY_FULL);

    for (i=0; i<Range; i++)
        (*StrBin) [i] = '0';
    (*StrBin) [i] = '\0';

    while (BInf <= Decim)
    {
        Exp++;
        BInf *= 2;
    }

    if (Exp)
    {
        Exp--;
        BInf = BInf / 2;
    }

    while (Exp >= 0)
    {
        if (Aux >= BInf)
        {
            Aux -= BInf;
            if (Exp < Range)
                (*StrBin) [Range-Exp-1] = '1';
        }
        else
        {
            if (Exp < Range)
                (*StrBin) [Range-Exp-1] = '0';
        }
    }

    Exp--;
    BInf = BInf / 2;
}
```

gnlstr.c

```
    return (GNL_OK);  
}
```

gnlsynt.c

```
/*-----*/  
/*  
 *      File:          gnlsynt.c  
 *      Version:        1.1  
 *      Modifications: -  
 *      Documentation: -  
 */  
/*-----*/  
/*      Class: none  
 *      Inheritance:  
 */  
/*-----*/  
  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlopt.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnloption.h"  
#include "time.h"  
#include "timecomp.e"  
  
/*-----*/  
/* GLOBAL VARIABLE */  
/*-----*/  
extern GNL_ENV      G_GnlEnv;  
extern BLIST        G_ListOfGnls;  
  
/*-----*/  
/* GnlSynthesize */  
/*-----*/  
GNL_STATUS GnlSynthesize ()  
{  
    int                  i;  
    GNL                  Gnl;  
    GNL                  GnlI;  
    BLIST                ListGnls;  
    LS_OPT_PANEL         OptPanel;  
    LIBC_LIB              GnlLib;  
    GNL_LIB               HGnlLib;  
    int                  Done;  
    int                  MapEffort;  
    FILE                 *OutFile;  
    GNL_STATUS            GnlStatus;  
    GNL                  TopGnl;  
    GNL_NETWORK           GlobalNetwork;  
    GNL_OPT_FORCE         Force;  
    GNL_CRITERION        Criter;  
    int                  Inject;  
    char                 *LibSourceFileName;  
  
    if (GnlEnvOutput ())  
    {
```

gnlsynt.c

```
if ((OutFile = fopen (GnlEnvOutput(), "w")) == NULL)
{
    fprintf (stderr, " ERROR: Cannot Open Output File '%s'\n",
             GnlEnvOutput());
    return (GNL_CANNOT_OPEN_OUTFILE);
}
fclose (OutFile);

if ((OutFile = fopen (GnlEnvLib(), "r")) == NULL)
{
    fprintf (stderr, " ERROR: Cannot Open Library File '%s'\n",
             GnlEnvLib());
    return (GNL_CANNOT_OPEN_LIBRARY);
}
fclose (OutFile);

if (GnlEnvInputFormat() == GNL_INPUT_VLG)
{
    fprintf (stderr, "\n Reading Verilog file [%s] ... \n",
             GnlEnvInput());
    if (GnlRead (GnlEnvInput(), &ListGnls))
    {
        fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
                 GnlEnvInput());
        return (GNL_CANNOT_OPEN_INPUTFILE);
    }
    fprintf (stderr, " Verilog Netlist Analyzed\n");
}
else
/* We read a .FSM format description. */ {
    if (BListCreate (&G_ListOfGnls))
        return (GNL_MEMORY_FULL);
    if ((GnlStatus = GnlFsmRead (GnlEnvInput(), &ListGnls)))
    {
        fprintf (stderr, " FSM file Analysis Aborted.\n");
        return (GnlStatus);
    }

    GnlUpdateComponentGnlDef (G_ListOfGnls);
}

/* Sort and prints the list 'ListGnls' by putting first the top      */
/* level modules                                                 */ SortTopLevelModules (ListGnls);
GnlPrintTopLevels (ListGnls);

if (!GnlEnvTop())
{
    GnlI = (GNL)BListElt (ListGnls, 0);
    TopGnl = GnlI;
    fprintf (stderr,
             " WARNING: Top Level Module is [%s] by default\n",
             GnlName (TopGnl));
}
else
```

```

{
    /* Taking the top level module from the user */ 
    TopGnl = NULL;
    for (i=0; i<BListSize (ListGnls); i++)
    {
        GnLI = (GNL)BListElt (ListGnls, i);
        if (!strcmp (GnlName (GnLI), GnlEnvTop()))
        {
            TopGnl = GnLI;
            break;
        }
    }
}

if (TopGnl == NULL)
{
    fprintf (stderr, " ERROR: cannot find top-level module [%s]\n",
             GnlEnvTop());
    return (GNL_CANNOT_FIND_TOP_LEVEL);
}

/* we create the Global network. */
if (GnlCreateNetwork (TopGnl, &GlobalNetwork))
    return (GNL_MEMORY_FULL);

/* Now we traverse the hierarchy in a top-down way and analyze      */
/* eventually black boxes in their corresponding files.           */
if (GnlEnvModelDir() && GnlEnvModelExt())
{
    fprintf (stderr, " Analyzing leaf cells...\n");
    if (GnlReadBlackBoxCells (GlobalNetwork, TopGnl, ListGnls,
                             GnlEnvModelDir(), GnlEnvModelExt()))
        return (GNL_MEMORY_FULL);
    fprintf (stderr, " \n");
}

/* Checking correctness of the network 'GlobalNetwork'.          */
if ((GnlStatus = GnlCheckNetwork (GlobalNetwork)))
    return (GnlStatus);

fprintf (stderr,
         "\n Checking Hierachical interfaces from top module [%s]\n",
         GnlName (TopGnl));

/* We verify the correct connection between user components and      */
/* their definitions and update some fields (RefCount).           */
if (GnlUpdateNCheckHierarchyInterface (GlobalNetwork, TopGnl,
                                       ListGnls))
    return (GNL_MEMORY_FULL);

/* Does the user invoke printing the hierarchy before flattening ? */
if ((GnlEnvPrintHierarchy () == GNL_PRINT_BEFORE_FLAT) ||
    (GnlEnvPrintHierarchy () == GNL_PRINT_BOTH))
{
    fprintf (stderr, "\n Hierarchy before Flattening:\n");
    fprintf (stderr, " -----\n");
}

```

```

        GnlPrintHierarchy (stderr, TopGnl, ListGnls);
    }

    if (GnlEnvFlatten() != GNL_NO_FLATTEN)
    {
        fprintf (stderr,
                  "\n Performing Flattening of modules in [%s]\n",
                  GnlName (TopGnl));
        if (GnlFlattenFullGnlComponents (GlobalNetwork, TopGnl, ListGnls))
            return (GNL_MEMORY_FULL);

        fprintf (stderr, " Flattening done\n");

        /* Checking correctness of the network 'GlobalNetwork'. */
        if ((GnlStatus = GnlCheckNetwork (GlobalNetwork)))
            return (GnlStatus);
    }

/* Does the user invoke printing the hierarchy after flattening ? */
if ((GnlEnvPrintHierarchy () == GNL_PRINT_AFTER_FLAT) ||
    (GnlEnvPrintHierarchy () == GNL_PRINT_BOTH))
{
    fprintf (stderr, "\n Hierarchy after Flattening:\n");
    fprintf (stderr, " -----\n");
    GnlPrintHierarchy (stderr, TopGnl, ListGnls);
}

/* Reading and building the technology library */
fprintf (stderr, "\n Reading Target Library [%s] ... \n",
         GnlEnvLib());
if (LibRead (GnlEnvLib(), &GnlLib))
{
    fprintf (stderr, " ERROR: Library analysis aborted\n");
    return (GNL_ERROR_LIBRARY_READING);
}
fprintf (stderr, " Library analyzed\n");

fprintf (stderr, "\n Building library elements... \n\n");
if ((GnlStatus = GnlBuildLib (GnlLib)))
    return (GnlStatus);

/* Setting the Optimization panel values */
Force = GnlEnvOptForce();
Criter = GnlEnvCriterion();
Inject = GnlEnvInject();
if ((GnlStatus = GnlCreateSynthesisPanel (Force, Criter, Inject,
                                         &OptPanel)))
    return (GnlStatus);

/* Synthesize the network 'GlobalNetwork'. */
HGNLlib = (GNL_LIB)LibHook (GnlLib);
if (GnlStrCopy (GnlEnvLib(), &LibSourceFileName))
    return (GNL_MEMORY_FULL);
SetGnlHLibSourceFileName (HGNLlib, LibSourceFileName);

/* We invoke the synthesis on this design */
if (GnlEnvOutput())

```

gnlsynt.c

```
{  
    if ((OutFile = fopen (GnlEnvOutput (), "w")) == NULL)  
    {  
        fprintf (stderr, " ERROR: Cannot Open Output File '%s'\n",  
                GnlEnvOutput ());  
        return (GNL_CANNOT_OPEN_OUTFILE);  
    }  
}  
  
fprintf (stderr, "\n");  
fprintf (stderr, " Linking Netlist Components With Libc Cells\n");  
if (GnlLinkLib (GlobalNetwork, TopGnl, GnlLib))  
    return (GNL_MEMORY_FULL);  
  
/* If the user asked to replace only combinational components ... */  
if (GnlEnvReplaceOnlyCombiCompo ())  
{  
    fprintf (stderr, " Replacing only Combinational components...\n");  
    if (GnlReplaceOnlyCombinatorialComponentsWithLinkLibCells (  
                    GlobalNetwork, TopGnl, GnlLib))  
        return (GNL_MEMORY_FULL);  
}  
else  
{  
    fprintf (stderr, " Replacing all components...\n");  
    if (GnlReplaceAllComponentsWithLinkLibCells (GlobalNetwork, TopGnl,  
                                                GnlLib))  
        return (GNL_MEMORY_FULL);  
}  
  
if ((GnlStatus = GnlSynthesizeNetwork (GlobalNetwork, GnlLib,  
                                         OutFile, OptPanel)))  
    return (GnlStatus);  
fclose (OutFile);  
  
/* We invoke the control fanout algorithm. */  
if (GnlEnvRespectLibConstraints ())  
{  
    if (GnlStatus = GnlMeetLibConstraints (GlobalNetwork, GnlLib))  
        return (GnlStatus);  
}  
  
/* We invoke the post optimisation on the synthesized design */  
if (GnlEnvPostOpt ())  
{  
    fprintf (stderr, " Performing Post-Optimization...\n");  
    if (TimeOptByResizing (GlobalNetwork, GnlLib))  
        return (GNL_MEMORY_FULL);  
    fprintf (stderr, "\n");  
  
#ifdef XXX_MODE  
    if ((GnlStatus = GnlPostOptimizeNetwork (GlobalNetwork, HGnlLib)))  
        return (GnlStatus);  
    fprintf (stderr, "\n");  
#endif  
}
```

gnlsynt.c

```
if (GnlReportDataAfterSynthesis (GlobalNetwork, GnlLib))
    return (GNL_MEMORY_FULL);

if (GnlPrintNetworkInfo (stderr, GlobalNetwork, GnlLib))
    return (GNL_MEMORY_FULL);

/* Printing the optimized mapped verilog netlist. */
fprintf (stderr, " Printing Verilog Netlist '%s'...\n\n",
         GnlEnvOutput ());
if ((GnlStatus = GnlPrintVerilogNetwork (GlobalNetwork, HGnlLib,
                                         GnlEnvOutput ())))
    return (GnlStatus);

fprintf (stderr, " Synthesis Done ! \n");
fprintf (stderr, " \n");

return (GNL_OK);
}

/*-----*/
```

gnltrans.c

```
/*-----*/  
/*-----*/  
/*      File:          gnltrans.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlopt.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnloption.h"  
  
/*-----*/  
/* GLOBAL VARIABLE */  
/*-----*/  
extern GNL_ENV    G_GnlEnv;  
extern BLIST     G_ListOfGnls;  
  
/*-----*/  
/* GnlSynthesizeAndEstimate */  
/*-----*/  
GNL_STATUS GnlTranslate ()  
{  
    BLIST           ListGnls;  
    GNL             TopGnl;  
    FILE            *OutFile;  
    GNL_STATUS      GnlStatus;  
  
    if (GnlEnvOutput ())  
    {  
        if ((OutFile = fopen (GnlEnvOutput (), "w")) == NULL)  
        {  
            fprintf (stderr, "# ERROR: Cannot Open Output File '%s'\n",  
                     GnlEnvOutput ());  
            return (GNL_CANNOT_OPEN_OUTFILE);  
        }  
        fclose (OutFile);  
    }  
  
    if (GnlEnvInputFormat () != GNL_INPUT_FSM)  
    {  
        fprintf (stderr, "# ERROR: translation is from 'fsm' format\n");  
        return (GNL_BAD_INPUT_FORMAT);  
    }  
else
```

gnltrans.c

```
/* We read a .FSM format description. */  
{  
    if (BListCreate (&G_ListOfGnls))  
        return (GNL_MEMORY_FULL);  
  
    if ((GnlStatus = GnlFsmRead (GnlEnvInput(), &ListGnls)))  
    {  
        fprintf (stderr, "# FSM file Analysis Aborted.\n");  
        return (GnlStatus);  
    }  
    fprintf (stderr, "# FSM file Analyzed\n");  
}  
  
TopGnl = (GNL)BListElt (ListGnls, 0);  
  
if ((OutFile = fopen (GnlEnvOutput(), "w")) == NULL)  
{  
    fprintf (stderr, "# ERROR: Cannot Open Output File '%s'\n",  
            GnlEnvOutput());  
    return (GNL_CANNOT_OPEN_OUTFILE);  
}  
  
GnlPrintGnl (OutFile, TopGnl);  
  
fclose (OutFile);  
  
return (GNL_OK);  
}  
/*-----*/
```

gnlverif.c

```
/*-----*/  
/*-----*/  
/*      File:          gnlverif.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY           /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "gnlopt.h"  
#include "bbdd.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
#include "gnloption.h"  
  
#include "blist.e"  
#include "libutil.e"  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern GNL_ENV    G_GnlEnv;  
extern BDD_PTR    GetBddPtrFromBdd ();  
extern LIBC_PIN   GnlGetPinCellWithName ();  
extern LIBC_LIB   G_GnlLibc;  
extern float     GnlGetCapaFromPin ();  
  
/*-----*/  
/* Global variables. */  
/*-----*/  
  
#ifdef O  
/*-----*/  
/* GnlPutComponentsInHashTable */  
/*-----*/  
#define LIST_COMPONENTS_HASH_TABLE_SIZE      1000  
GNL_STATUS GnlPutComponentsInHashTable (Components,  
                                         HashListPredefCompo,  
                                         HashListUserCompo)  
BLIST    Components;  
BLIST    *HashListPredefCompo;  
BLIST    *HashListUserCompo;  
{  
    int             i;  
    BLIST          NewList;
```

gnlverif.c

```
GNL_COMPONENT ComponentI;
GNL_USER_COMPONENT UserCompo;
unsigned int Key;
unsigned int Key1;
unsigned int Key2;
char *InstanceName;
BLIST Bucket;

if (BListCreateWithSize (LIST_COMPONENTS_HASH_TABLE_SIZE,
                        HashListPredefCompo))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (LIST_COMPONENTS_HASH_TABLE_SIZE,
                        HashListUserCompo))
    return (GNL_MEMORY_FULL);

for (i=0; i<LIST_COMPONENTS_HASH_TABLE_SIZE; i++)
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (HashListPredefCompo, (int)NewList))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (HashListUserCompo, (int)NewList))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
    {
        UserCompo = (GNL_USER_COMPONENT)ComponentI;

        /* The 'Key' of the component is a function of its name */
        /* and instance name */
        Key1 = KeyOfName (GnlUserComponentName (UserCompo),
                          LIST_COMPONENTS_HASH_TABLE_SIZE);
        Key2 = KeyOfName (GnlUserComponentInstName (UserCompo),
                          LIST_COMPONENTS_HASH_TABLE_SIZE);
        Key = (Key1+Key2) % LIST_COMPONENTS_HASH_TABLE_SIZE;

        Bucket = (BLIST)BListElt (HashListUserCompo, Key);
        if (BListAddElt (Bucket, (int)ComponentI))
            return (GNL_MEMORY_FULL);
        continue;
    }

    /* otherwise it is a predefined component */
    if (GnlComponentType (ComponentI) == GNL_TRISTATE_COMPO)
    {
        InstanceName = GnlTriStateInstName (ComponentI);
    }
    if (GnlComponentType (ComponentI) == GNL_SEQUENTIAL_COMPO)
    {
        InstanceName = GnlUserComponentInstName (ComponentI);
    }
}
```

gnlverif.c

```
        }
        Key = KeyOfName (InstanceName,
                          LIST_COMPONENTS_HASH_TABLE_SIZE);
        Bucket = (BLIST)BlistElt (HashTablePredefCompo, Key);
        if (BListAddElt (Bucket, (int)ComponentI))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* GnlVerifNetworkRec */
/*-----*/
GNL_STATUS GnlVerifNetworkRec (Nw1, Nw2, Gn11, Gn12, GnlLibc)
    GNL_NETWORK  Nw1;
    GNL_NETWORK  Nw2;
    GNL          *Gn11;
    GNL          *Gn12;
    LIBC_LIB GnlLibc;
{
    int             i;
    GNL_COMPONENT  ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL          GnlCompoI;
    GNL          OptGnl;
    BLIST         Components;
    int           Done;
    BLIST         ListAssoc;

    if (GnlTag (Gn11) == GnlNetworkTag (Nw1))
        return (GNL_OK);

    SetGnlTag (Gn11, GnlNetworkTag (Nw1));
    SetGnlTag (Gn12, GnlNetworkTag (Nw2));

    Components1 = GnlComponents (Gn11);
    Components2 = GnlComponents (Gn12);

    if (BListCreate (&ListAssoc))
        return (GNL_MEMORY_FULL);

    if (BListSize (Components1) != BListSize (Components2))
    {
        fprintf (stderr,
 "# WARNING: Modules <%s> and <%s> do not have the same number of
components\n",
                GnlName (Gn11), GnlName (Gn12));
        return (GNL_OK);
    }

    if (GnlPutComponentsInHashTable (Components1, &PredefCompoHashTable1,
                                    &UserCompoHashTable1))
        return (GNL_MEMORY_FULL);
```

gnlverif.c

```
/* we scan components of the second netlist and try to find the      */
/* equivalent components in the first one.                           */
for (i=0; i<BListSize (Components2); i++)
{
    Found = 0;
    Component2I = (GNL_COMPONENT)BListElt (Components2, i);

    if (GnlComponentType (Component2I) == GNL_USER_COMPO)
    {
        /* We look for component of same name and instance name      */
        Component1I = GnlLookForSameComponent (UserCompoHashList1);
        if (!Component1I)
        {
            fprintf (stderr,
"# WARNING: user instance <%s> in module [%s] has no
corresponding instance in module [%s]\n",
                GnlUserComponentInstName ((GNL_USER_COMPONENT)
                                         Component2I),
                GnlName (Gnl2), GnlName (Gnl1));
            return (GNL_OK);
        }

        /* we call the equivalence between the two instances   */
        /* to do ...                                           */
        continue;
    }

    /* Case of a predefined component.                         */
    Component1I = GnlLookForSameComponent (PredefCompoHashList1);

    /* we consider that the two components use the same parameters */
    /* ex: toto u1 (a, b) and toto u1 (x, y) means that the */
    /* variables 'a' <-> 'x' and 'b' <-> 'y' are tied       */
    if (GnlTiedInstanceParameters (Component2I, Component1I,
                                   ListAssoc))
        return (GNL_MEMORY_FULL);

    /* We call the real boolean function equivalence between functions */
    /* of 'Gnl1' and 'Gnl2'. Moreover we use the 'ListAssoc' as      */
    /* correspondance between the variables of 'Gnl1' and 'Gnl2'. */
    return (GNL_OK);
}

/*-----*/
/* GnlVerifNetworks                                         */
/*-----*/
/* Main EC verification procedure which verifies that both networks */
/* 'Nw1' and 'Nw2' are equivalent.                            */
/*-----*/
GNL_STATUS GnlVerifNetworks (Nw1, Nw2, GnlLibc)
```

gnlverif.c

```
GNL_NETWORK      Nw1;
GNL_NETWORK      Nw2;
LIBC_LIB GnlLibc;
{
    GNL          TopGn11;
    GNL          TopGn12;
    GNL_LIB     HGnlLib;
    int         MaxTag;

    HGnlLib = (GNL_LIB)LibHook (GnlLibc);

    MaxTag = (GnlNetworkTag (Nw1) > GnlNetworkTag (Nw2) ?
               GnlNetworkTag (Nw1) : GnlNetworkTag (Nw2));
    SetGnlNetworkTag (Nw1, MaxTag+1);
    SetGnlNetworkTag (Nw2, MaxTag+1);

    TopGn11 = GnlNetworkTopGnl (Nw1);
    TopGn12 = GnlNetworkTopGnl (Nw2);
    if (GnlVerifNetworksRec (Nw1, Nw2, TopGn11, TopGn12, GnlLibc))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

#endif

/*-----*/
/* GnlGetActualVarOfFormalName */
/*-----*/
/* This procedure returns the Actual signal associated to the formal */
/* port of name 'VarName'. */
/* ASSUMPTION: the formal port has a char * type and the actual port is */
/* GNL_VAR type (not of type GNL_NODE). */
/*-----*/
int GnlGetActualVarOfFormalName (Interface, VarName, Actual)
    BLIST      Interface;
    char       *VarName;
    GNL_VAR   *Actual;
{
    int           i;
    GNL_ASSOC    AssocI;
    char       *Formal;

    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        Formal = GnlAssocFormalPort (AssocI);

        if (!Formal)
            continue;

        *Actual = GnlAssocActualPort (AssocI);

        if (!strcmp (Formal, VarName))
```

gnlverif.c

```
        return (1);
    }

    *Actual = NULL;
    return (0);
}

/*-----
/* GntGetClkSeqVar
/*-----
/* Returns the Clock Var associated to Clock pin of the cell.
/*-----*/
GNL_VAR GntGetClkSeqVar (UserCompo, Cell, Interface, Clock, Name)
    GNL_USER_COMPONENT UserCompo;
    LIBC_CELL Cell;
    BLIST Interface;
    LIBC_BOOL_OPR *Clock;
    char **Name;
{
    LIBC_PIN Pins;
    LIBC_NAME_LIST ListPinName;
    GNL_VAR Var;
    LIBC_FF_LATCH FFLatch;

    FFLatch = LibCellFFLatch (Cell);

    if (LibFFLatchIsFF (FFLatch))
        *Clock = LibFFLatchClockOn (FFLatch);
    else
        *Clock = LibFFLatchEnable (FFLatch);

    *Name = NULL;
    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simple name */
        if (GnlNameIsPresent (LibNameListName (ListPinName), *Clock))
        {
            *Name = LibNameListName (ListPinName);
            break;
        }
    }

    if (!(*Name))
    {
        fprintf (stderr,
                 " ERROR: cannot find clock pin in <%s>\n",
                 GnlUserComponentInstName (UserCompo));
        exit (1);
    }

    if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))
    {
        fprintf (stderr,
                 " ERROR: cannot find actual parameter associated to clock pin <%s> in
<%s>\n",
                 GnlUserComponentInstName (UserCompo),
                 *Name);
    }
}
```

```

        *Name, GnlUserComponentInstName (UserCompo));
    exit (1);
}

return (Var);
}

/*-----
/* GntGetRstSeqVar
/*-----
/* Returns the Reset Var associated to Clear pin of the cell.
/*-----
GNL_VAR GntGetRstSeqVar (UserCompo, Cell, Interface, Clear, Name)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL              Cell;
    BLIST                 Interface;
    LIBC_BOOL_OPR          *Clear;
    char                  **Name;
{
    LIBC_PIN                Pins;
    LIBC_NAME_LIST           ListPinName;
    GNL_VAR                 Var;
    LIBC_FF_LATCH            FFLatch;

    FFLatch = LibCellFFLatch (Cell);

    *Clear = LibFFLatchClear (FFLatch);

    if (!(*Clear))
        return (NULL);

    *Name = NULL;
    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simple name */
        if (GnlNameIsPresent (LibNameListName (ListPinName), *Clear))
        {
            *Name = LibNameListName (ListPinName);
            break;
        }
    }

    if (!(*Name))
    {
        fprintf (stderr,
                 " ERROR: cannot find Clear pin in <%s>\n",
                 GnlUserComponentInstName (UserCompo));
        exit (1);
    }

    if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))
    {
        fprintf (stderr,
                 " ERROR: cannot find actual parameter associated to Clear pin <%s> in
<%s>\n",

```

gnlverif.c

```
        *Name, GnlUserComponentInstName (UserCompo));
    exit (1);
}

return (Var);
}

/*-----
/* GntGetSetSeqVar
/*-----
/* Returns the Set Var associated to Preset pin of the cell.
/*-----*/
GNL_VAR GntGetSetSeqVar (UserCompo, Cell, Interface, Preset, Name)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL              Cell;
    BLIST                 Interface;
    LIBC_BOOL_OPR   *Preset;
    char                  **Name;
{
    LIBC_PIN                Pins;
    LIBC_NAME_LIST           ListPinName;
    GNL_VAR                 Var;
    LIBC_FF_LATCH            FFLatch;

FFLatch = LibCellFFLatch (Cell);

*Preset = LibFFLatchPreset (FFLatch);

if (!(*Preset))
    return (NULL);

*Name = NULL;
Pins = LibCellPins (Cell);
for (;Pins != NULL; Pins = LibPinNext(Pins))
{
    ListPinName = LibPinName (Pins); /* we know it is a simple name */
    if (GnlNameIsPresent (LibNameListName (ListPinName), *Preset))
    {
        *Name = LibNameListName (ListPinName);
        break;
    }
}

if (!(*Name))
{
    fprintf (stderr,
        " ERROR: cannot find Preset pin in <%s>\n",
        GnlUserComponentInstName (UserCompo));
    exit (1);
}

if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))
{
    fprintf (stderr,
    " ERROR: cannot find actual parameter associated to Preset pin <%s> in
<%s>\n",
```

gnlverif.c

```
        *Name, GnlUserComponentInstName (UserCompo));
    exit (1);
}

return (Var);
}

/*-----
/* GntGetInputSeqVar
/* -----
/* Returns the Input Var associated to Preset pin of the cell.
/* -----
GNL_VAR GntGetInputSeqVar (UserCompo, Cell, Interface, Input, Name)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL              Cell;
    BLIST                 Interface;
    LIBC_BOOL_OPR          *Input;
    char                  **Name;
{
    LIBC_PIN                Pins;
    LIBC_NAME_LIST           ListPinName;
    GNL_VAR                 Var;
    LIBC_FF_LATCH            FFLatch;

    FFLatch = LibCellFFLatch (Cell);

    if (LibFFLatchIsFF (FFLatch))
        *Input = LibFFLatchNextState (FFLatch);
    else
        *Input = LibFFLatchDataIn (FFLatch);

    *Name = NULL;
    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simple name */
        if (GnlNameIsPresent (LibNameListName (ListPinName), *Input))
        {
            *Name = LibNameListName (ListPinName);
            break;
        }
    }

    if (!(*Name))
    {
        fprintf (stderr,
            " ERROR: cannot find Input pin in <%s>\n",
            GnlUserComponentInstName (UserCompo));
        exit (1);
    }

    if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))
    {
        fprintf (stderr,
        " ERROR: cannot find actual parameter associated to Input pin <%s> in
<%s>\n",

```

gnlverif.c

```
        *Name, GnlUserComponentInstName (UserCompo));
    exit (1);
}

return (Var);
}

/*-----
/* GntGetCellOutputQSeqVar
/*-----*/
/* Returns the output Var Q associated to output pin Q of the sequential*/
/* cell. */
/*-----*/
GNL_VAR GntGetCellOutputQSeqVar (UserCompo, Cell, Interface, Name)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL              Cell;
    BLIST                  Interface;
    char                   **Name;
{
    LIBC_PIN                Pins;
    LIBC_PIN                OutputPin;
    LIBC_BOOL_OPR           PinFunction;
    GNL_NODE                Node;
    char                    *PinName;
    GNL_VAR                 OutVar;
    GNL_FUNCTION             NewFunction;
    LIBC_NAME_LIST           ListPinName;
    char                    *IQ;

    IQ = LibFFLatchQName (LibCellFFLatch (Cell));

    Pins = LibCellPins (Cell);

    *Name = NULL;
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins);

        /* 'PinName' can be a bundle a bus or a pin group */
        if (!GnlSinglePinName (ListPinName))
        {
            fprintf (stderr,
                    " WARNING: Cell <%s> has a complex output pin name\n",
                    LibCellName (Cell));
            exit (1);
        }

        PinName = LibNameListName (ListPinName);
        PinFunction = LibPinFunction (Pins);
        if (PinFunction)
        {
            if (GnlNameIsPresent (IQ, PinFunction))
            {
                *Name = PinName;
                break;
            }
        }
    }
}
```

```

        }
    }

    if (!(*Name))
        return (NULL);

    if (!GnlGetActualVarOfFormalName (Interface, *Name, &OutVar))
    {
        fprintf (stderr,
        " WARNING: cannot find output pin <%s> in Cell <%s>: Port is open.\n",
        *Name,
        LibCellName (Cell));
        return (NULL);
    }

    return (OutVar);
}

/*-----
/* GntGetCellOutputQBarSeqVar
/*-----
/* Returns the output Var Q Bar associated to output pin Q Bar of the   */
/* sequential cell.                                                 */
/*-----*/
GNL_VAR GntGetCellOutputQBarSeqVar (UserCompo, Cell, Interface, Name)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL             Cell;
    BLIST                 Interface;
    char                  **Name;
{
    LIBC_PIN               Pins;
    LIBC_PIN               OutputPin;
    LIBC_BOOL_OPR          PinFunction;
    GNL_NODE               Node;
    char                   *PinName;
    GNL_VAR                OutVar;
    GNL_FUNCTION           NewFunction;
    LIBC_NAME_LIST          ListPinName;
    char                   *IQN;

    IQN = LibFFLatchQName (LibCellFFLatch (Cell));

    Pins = LibCellPins (Cell);

    *Name = NULL;
    for (;Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins);

        /* 'PinName' can be a bundle a bus or a pin group           */
        if (!GnlSinglePinName (ListPinName))                      */
        {
            fprintf (stderr,
            " WARNING: Cell <%s> has a complex output pin name\n",
            LibCellName (Cell));
        }
    }
}

```

```

        exit (1);
    }

    PinName = LibNameListName (ListPinName);
    PinFunction = LibPinFunction (Pins);
    if (PinFunction)
    {
        if (GnlNameIsPresent (IQN, PinFunction))
        {
            *Name = PinName;
            break;
        }
    }
}

if (!(*Name))
    return (NULL);

if (!GnlGetActualVarOfFormalName (Interface, *Name, &OutVar))
{
    fprintf (stderr,
    " WARNING: cannot find output pin <%s> in Cell <%s>: Port is open.\n",
    *Name,
    LibCellName (Cell));
    return (NULL);
}

return (OutVar);
}

/*
/* GntGetCellOutputVar
 */
/*
/* Returns the output Var associated to output pin of the cell. We      */
/* pick up the first pin if several output pins.                      */
/*
GNL_VAR GntGetCellOutputVar (UserCompo, Cell, Interface, PinName)
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL             Cell;
    BLIST                 Interface;
    char                  **PinName;
{
    LIBC_PIN              Pins;
    LIBC_PIN              OutputPin;
    LIBC_BOOL_OPR         Function;
    GNL_NODE              Node;
    GNL_VAR               OutVar;
    GNL_FUNCTION          NewFunction;
    LIBC_NAME_LIST         ListPinName;

    Pins = LibCellPins (Cell);

    /* we extract the first output Pin. If no output pin then we return */
    GnlGetOutputPinFromPins (Pins, &OutputPin);
    if (!OutputPin)

```

gnlverif.c

```
{  
    fprintf (stderr, " WARNING: Cell <%s> has no output pin\n",
              LibCellName (Cell));  
  
    return (GNL_OK);
}  
  
ListPinName = LibPinName (OutputPin);  
  
/* Since 'PinName' can be a bundle a bus or a pin group we need to */  
if (!GnlSinglePinName (ListPinName))  
{  
    fprintf (stderr,
              " WARNING: Cell <%s> has a complex output pin name\n",
              LibCellName (Cell));  
    exit (1);
}  
  
*PinName = LibNameListName (ListPinName);  
  
if (!GnlGetActualVarOfFormalName (Interface, *PinName, &OutVar))  
{  
    fprintf (stderr,
              " ERROR: cannot find output pin <%s> in Cell <%s>\n",
              *PinName,
              LibCellName (Cell));
    exit (1);
}  
  
return (OutVar);
}  
  
/*-----*/  
/* GnlModifyComponentIntoSeqComponent */  
/*-----*/  
/* This procedure destroys the GNL_USER_COMPONENT 'UserCompo' and */  
/* replaces it by a GNL_SEQUENTIAL_COMPONENT representing the same */  
/* interface thru 'SeqCompo'. */  
/*-----*/  
GNL_STATUS GnlModifyComponentIntoSeqComponent (Gnl, UserCompo, GnlLib,
                                                ReplaceCombi, SeqCompo)  
{  
    GNL                      Gnl;  
    GNL_USER_COMPONENT        UserCompo;  
    LIBC_LIB                  GnlLib;  
    int                      ReplaceCombi;  
    GNL_SEQUENTIAL_COMPONENT *SeqCompo;  
  
    BLIST                     UserCompoInterface;  
    GNL_VAR                   OutVar;  
    GNL_VAR                   OutVarBar;  
    GNL_VAR                   ClkVar;  
    GNL_VAR                   RstVar;  
    GNL_VAR                   SetVar;  
    GNL_VAR                   InputVar;  
    LIBC_CELL                 Cell;  
    char                      *NewName;  
    LIBC_FF_LATCH             FFLatch;
```

gnlverif.c

```
char *FormalNameOutQ;
char *FormalNameOutQBar;
char *FormalNameClkVar;
char *FormalNameRstVar;
char *FormalNameSetVar;
char *FormalNameInputVar;
GNL_ASSOC Assoc;
GNL_SEQUENTIAL_COMPO_INFO NewSeqCompoInfo;
LIBC_BOOL_OPR InputOpr;
LIBC_BOOL_OPR ClockOpr;
LIBC_BOOL_OPR ResetOpr;
LIBC_BOOL_OPR SetOpr;

UserCompoInterface = GnlUserComponentInterface (UserCompo);
Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompo);

OutVar = GntGetCellOutputQSeqVar (UserCompo, Cell, UserCompoInterface,
&FormalNameOutQ);
OutVarBar = GntGetCellOutputQBarSeqVar (UserCompo, Cell,
UserCompoInterface,
&FormalNameOutQBar);
ClkVar = GntGetClkSeqVar (UserCompo, Cell, UserCompoInterface,
&ClockOpr, &FormalNameClkVar);
RstVar = GntGetRstSeqVar (UserCompo, Cell, UserCompoInterface,
&ResetOpr, &FormalNameRstVar);
SetVar = GntGetSetSeqVar (UserCompo, Cell, UserCompoInterface,
&SetOpr, &FormalNameSetVar);
InputVar = GntGetInputSeqVar (UserCompo, Cell, UserCompoInterface,
&InputOpr, &FormalNameInputVar);

FFLatch = LibCellFFLatch (Cell);

/* Is it a DFF or a LATCH ? */
if (LibFFLatchIsFF (FFLatch))
{
    if (GnlCreateSequentialComponent (GNL_DFF, SeqCompo))
        return (GNL_MEMORY_FULL);
}
else
{
    if (GnlCreateSequentialComponent (GNL_LATCH, SeqCompo))
        return (GNL_MEMORY_FULL);
}

SetGnlSequentialCompoInput (*SeqCompo, InputVar);
Assoc = GnlSequentialCompoInputAssoc (*SeqCompo);
SetGnlAssocFormalPort (Assoc, FormalNameInputVar);

if (OutVar)
{
    SetGnlSequentialCompoOutput (*SeqCompo, OutVar);
    Assoc = GnlSequentialCompoOutputAssoc (*SeqCompo);
    SetGnlAssocFormalPort (Assoc, FormalNameOutQ);
}

if (OutVarBar)
```

```

{
    SetGnlSequentialCompoOutputBar (*SeqCompo, OutVarBar);
    Assoc = GnlSequentialCompoOutputBarAssoc (*SeqCompo);
    SetGnlAssocFormalPort (Assoc, FormalNameOutQBar);
}

SetGnlSequentialCompoClock (*SeqCompo, ClkVar);
Assoc = GnlSequentialCompoClockAssoc (*SeqCompo);
SetGnlAssocFormalPort (Assoc, FormalNameClkVar);
if (GnlBoolOprIsSimpleSignal (ClockOpr))
    SetGnlSequentialCompoClockPol (*SeqCompo, 1);
else
    SetGnlSequentialCompoClockPol (*SeqCompo, 0);

SetGnlSequentialCompoReset (*SeqCompo, RstVar);
Assoc = GnlSequentialCompoResetAssoc (*SeqCompo);
SetGnlAssocFormalPort (Assoc, FormalNameRstVar);
if (GnlBoolOprIsSimpleSignal (ResetOpr))
    SetGnlSequentialCompoResetPol (*SeqCompo, 1);
else
    SetGnlSequentialCompoResetPol (*SeqCompo, 0);

SetGnlSequentialCompoSet (*SeqCompo, SetVar);
Assoc = GnlSequentialCompoSetAssoc (*SeqCompo);
SetGnlAssocFormalPort (Assoc, FormalNameSetVar);
if (GnlBoolOprIsSimpleSignal (SetOpr))
    SetGnlSequentialCompoSetPol (*SeqCompo, 1);
else
    SetGnlSequentialCompoSetPol (*SeqCompo, 0);

if (GnlStrCopy (GnlUserComponentInstName (UserCompo), &NewName))
    return (GNL_MEMORY_FULL);

SetGnlSequentialCompoInstName (*SeqCompo, NewName);

if (!ReplaceCombi)
{
    if (GnlCreateSequentialCompoInfo (&NewSeqCompoInfo))
        return (GNL_MEMORY_FULL);
    SetGnlSequentialCompoHook (*SeqCompo, NewSeqCompoInfo);
    SetGnlSeqCompoInfoCell (*SeqCompo, Cell);

    SetGnlSeqCompoInfoOutput (*SeqCompo, FormalNameOutQ);
    SetGnlSeqCompoInfoOutputBar (*SeqCompo, FormalNameOutQBar);

    SetGnlSeqCompoInfoInput (*SeqCompo, InputOpr);
    SetGnlSeqCompoInfoClock (*SeqCompo, ClockOpr);
    SetGnlSeqCompoInfoReset (*SeqCompo, ResetOpr);
    SetGnlSeqCompoInfoSet (*SeqCompo, SetOpr);
}

GnlFreeUserComponent (UserCompo);

return (GNL_OK);
}

/*-----*/

```

gnlverif.c

```
/* GnlGetInputTristateVar */  
/*-----*/  
/* Returns the Input Var associated to Input pin of the cell. */  
/*-----*/  
GNL_VAR GnlGetInputTristateVar (UserCompo, Cell, Interface, Function,  
                                Name)  
{  
    GNL_USER_COMPONENT    UserCompo;  
    LIBC_CELL              Cell;  
    BLIST                 Interface;  
    LIBC_BOOL_OPR   *Function;  
    char                  **Name;  
    LIBC_PIN               Pins;  
    LIBC_NAME_LIST          ListPinName;  
    GNL_VAR                Var;  
    LIBC_PIN               OutPin;  
  
    if (!LibCellWith3StatePin (Cell, &OutPin))  
    {  
        fprintf (stderr,  
                  " ERROR: cannot find output pin of tri-state <%s>\n",
                  LibCellName (Cell));  
        exit (1);  
    }  
  
    *Function = LibPinFunction (OutPin);  
  
    *Name = NULL;  
    Pins = LibCellPins (Cell);  
    for (;Pins != NULL; Pins = LibPinNext(Pins))  
    {  
        ListPinName = LibPinName (Pins); /* we know it is a simple name */  
        if (GnlNameIsPresent (LibNameListName (ListPinName), *Function))  
        {  
            *Name = LibNameListName (ListPinName);  
            break;  
        }  
    }  
  
    if (!(*Name))  
    {  
        fprintf (stderr,  
                  " ERROR: cannot find Input pin in tri-state <%s>\n",
                  GnlUserComponentInstName (UserCompo));  
        exit (1);  
    }  
  
    if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))  
    {  
        fprintf (stderr,  
                  " ERROR: cannot find actual parameter associated to Input pin <%s> in  

<%s>\n",
                  *Name, GnlUserComponentInstName (UserCompo));  
        exit (1);  
    }
```

gnlverif.c

```
    return (Var);
}

/*-----
/* GnlGetEnableTristateVar
/*-----*/
/* Returns the Enable Var associated to Enable pin of the cell. */
/*-----*/
GNL_VAR GnlGetEnableTristateVar (UserCompo, Cell, Interface, ThreeState,
                                Name)
{
    GNL_USER_COMPONENT    UserCompo;
    LIBC_CELL              Cell;
    BLIST                  Interface;
    LIBC_BOOL_OPR   *ThreeState;
    char                  **Name;
{
    LIBC_PIN                Pins;
    LIBC_NAME_LIST          ListPinName;
    GNL_VAR                 Var;
    LIBC_PIN                OutPin;

    if (!LibCellWith3StatePin (Cell, &OutPin))
    {
        fprintf (stderr,
                  " ERROR: cannot find output pin of tri-state <%s>\n",
                  LibCellName (Cell));
        exit (1);
    }

    *ThreeState = LibPin3State (OutPin);

    *Name = NULL;
    Pins = LibCellPins (Cell);
    for (; Pins != NULL; Pins = LibPinNext(Pins))
    {
        ListPinName = LibPinName (Pins); /* we know it is a simple name */
        if (GnlNameIsPresent (LibNameListName (ListPinName), *ThreeState))
        {
            *Name = LibNameListName (ListPinName);
            break;
        }
    }

    if (!(*Name))
    {
        fprintf (stderr,
                  " ERROR: cannot find Enable pin in tri-state <%s>\n",
                  GnlUserComponentInstName (UserCompo));
        exit (1);
    }

    if (!GnlGetActualVarOfFormalName (Interface, *Name, &Var))
    {
        fprintf (stderr,
                  " ERROR: cannot find actual parameter associated to Enable pin <%s> in
<%s>\n",

```

```

        *Name, GnlUserComponentInstName (UserCompo));
    exit (1);
}

return (Var);
}

/*
 *-----*/
/* GnlModifyComponentIntoTristateComponent */
/*-----*/
/* This procedure destroys the GNL_USER_COMPONENT 'UserCompo' and */
/* replaces it by a GNL_TRISTATE_COMPONENT representing the same */
/* interface thru 'TristateCompo'. */
/*-----*/
GNL_STATUS GnlModifyComponentIntoTristateComponent (Gnl, UserCompo,
                                                    GnlLib, ReplaceCombi, TristateCompo)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT         UserCompo;
    LIBC_LIB                  GnlLib;
    int                       ReplaceCombi;
    GNL_TRISTATE_COMPONENT    *TristateCompo;

    BLIST                     UserCompoInterface;
    char                      *NewName;
    LIBC_CELL                 Cell;
    GNL_VAR                   OutVar;
    GNL_VAR                   InputVar;
    GNL_VAR                   EnableVar;
    char                      *FormalNameOutVar;
    char                      *FormalNameInputVar;
    char                      *FormalNameEnableVar;
    GNL_ASSOC                 Assoc;
    GNL_TRISTATE_COMPO_INFO   NewTriStateCompoInfo;
    LIBC_BOOL_OPR              InputOpr;
    LIBC_BOOL_OPR              EnableOpr;

    UserCompoInterface = GnlUserComponentInterface (UserCompo);
    Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompo);

    if (GnlCreateTristateComponent (TristateCompo))
        return (GNL_MEMORY_FULL);

    if (BListSize (UserCompoInterface) != 3)
    {
        fprintf (stderr,
                 " ERROR: Tri-state <%s> has a wrong number of parameters\n",
                 GnlUserComponentInstName (UserCompo));
        exit (1);
    }

    OutVar = GntGetCellOutputVar (UserCompo, Cell, UserCompoInterface,
                                 &FormalNameOutVar);
    InputVar = GnlGetInputTristateVar (UserCompo, Cell,
                                       UserCompoInterface, &InputOpr,
                                       &FormalNameInputVar);
}

```

```

EnableVar = GnlGetEnableTristateVar (UserCompo, Cell,
                                    UserCompoInterface, &EnableOpr,
                                    &FormalNameEnableVar);

SetGnlTriStateInput (*TristateCompo, InputVar);
Assoc = GnlTriStateInputAssoc (*TristateCompo);
SetGnlAssocFormalPort (Assoc, FormalNameInputVar);
if (GnlBoolOprIsSimpleSignal (InputOpr))
    SetGnlTriStateInputPol (*TristateCompo, 1);
else
    SetGnlTriStateInputPol (*TristateCompo, 0);

SetGnlTriStateOutput (*TristateCompo, OutVar);
Assoc = GnlTriStateOutputAssoc (*TristateCompo);
SetGnlAssocFormalPort (Assoc, FormalNameOutVar);

SetGnlTriStateSelect (*TristateCompo, EnableVar);
Assoc = GnlTriStateSelectAssoc (*TristateCompo);
SetGnlAssocFormalPort (Assoc, FormalNameEnableVar);
if (GnlBoolOprIsSimpleSignal (EnableOpr))
    SetGnlTriStateSelectPol (*TristateCompo, 1);
else
    SetGnlTriStateSelectPol (*TristateCompo, 0);

if (GnlStrCopy (GnlUserComponentInstName (UserCompo), &NewName))
    return (GNL_MEMORY_FULL);

SetGnlTriStateInstName (*TristateCompo, NewName);

if (!ReplaceCombi)
{
    if (GnlCreateTriStateCompoInfo (&NewTriStateCompoInfo))
        return (GNL_MEMORY_FULL);
    SetGnlTriStateHook (*TristateCompo, NewTriStateCompoInfo);
    SetGnlTriStateCompoInfoCell (*TristateCompo, Cell);
    SetGnlTriStateCompoInfoOutput (*TristateCompo, FormalNameOutVar);
    SetGnlTriStateCompoInfoInput (*TristateCompo, InputOpr);
    SetGnlTriStateCompoInfoEnable (*TristateCompo, EnableOpr);
}

GnlFreeUserComponent (UserCompo);

return (GNL_OK);
}

/*
 * GnlGetGnlNodeFromBoolOprWithInterface
 */
/*
 * This procedure returns the GNL_NODE expression thru 'GnlNode' of the */
/* original tree expression of type LIBC_BOOL_OPR generated in the LIBC */
/* parsing. Indexsignal, buses are not considered and we exit(1). */
/* The GNL_NODE objects are created in the memory segment of 'Gnl'. */
*/
GNL_STATUS GnlGetGnlNodeFromBoolOprWithInterface (Gnl, CompoName,
                                                Interface, BoolOpr, GnlNode)

```

```

GNL          Gnl;
char         *CompoName;
BLIST        Interface;
LIBC_BOOL_OPR BoolOpr;
GNL_NODE     *GnlNode;

{
    text_buffer      *VarName;
    GNL_STATUS       GnlStatus;
    GNL_VAR          GnlVar;
    GNL_NODE         GnlNodeLeft;
    GNL_NODE         GnlNodeRight;
    BLIST           NewList;

/* terminal bool opr. */
switch (LibBoolOprType (BoolOpr)) {
    /* terminal case of a signal. */
    case ID_B:
        VarName = LibBoolOprIdName (BoolOpr);
        if (!GnlGetActualVarOfFormalName (Interface, VarName, &GnlVar))
        {
            fprintf (stderr,
                      " ERROR: Cannot find signal <%s> in interface of '%s'\n",
                      VarName, CompoName);
            exit (1);
        }
        if (GnlCreateNode (Gnl, GNL_VARIABLE, GnlNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*GnlNode, (BLIST)GnlVar);
        break;

    case ZERO_B:
        if (GnlCreateNodeVss (Gnl, GnlNode))
            return (GNL_MEMORY_FULL);
        break;

    case ONE_B:
        if (GnlCreateNodeVdd (Gnl, GnlNode))
            return (GNL_MEMORY_FULL);
        break;

/* case of binary Boolean operators */
    case XOR_B:
    case OR_B:
    case AND_B:
        if (!GnlGetGnlNodeFromBoolOprWithInterface (Gnl, CompoName,
                                                    Interface,
                                                    LibBoolOprLeftSon (BoolOpr),
                                                    &GnlNodeLeft))
            return (GNL_MEMORY_FULL);
        if (!GnlGetGnlNodeFromBoolOprWithInterface (Gnl, CompoName,
                                                    Interface,
                                                    LibBoolOprRightSon (BoolOpr),
                                                    &GnlNodeRight))
            return (GNL_MEMORY_FULL);

        if (LibBoolOprType (BoolOpr) == XOR_B)

```

```

{
    if (GnlCreateNodeXor (Gnl, GnlNodeLeft, GnlNodeRight,
                          GnlNode, 0))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

if (BListCreateWithSize (2, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddEl (NewList, (int)GnlNodeLeft))
    return (GNL_MEMORY_FULL);
if (BListAddEl (NewList, (int)GnlNodeRight))
    return (GNL_MEMORY_FULL);

if (GnlCreateNode (Gnl,
                  GnlOpFromBoolOprProp (LibBoolOprType (BoolOpr)),
                  GnlNode))
    return (GNL_MEMORY_FULL);
SetGnlNodeSons (*GnlNode, NewList);
break;

/* Unary Boolean operators. It son is the right one */
case NOT_B:
    if (GnlGetGnlNodeFromBoolOprWithInterface (Gnl, CompoName,
                                                Interface,
                                                LibBoolOprRightSon (BoolOpr),
                                                &GnlNodeRight))
        return (GNL_MEMORY_FULL);
    if (GnlCreateNodeNot (Gnl, GnlNodeRight, GnlNode))
        return (GNL_MEMORY_FULL);
    break;

default:
    fprintf (stderr,
"Operator unsupported in 'GnlGetGnlNodeFromBoolOprWithInterface'\n");
    exit (1);
}

return (GNL_OK);
}

/*
 * GnlModifyComponentIntoLogicFunction
 */
GNL_STATUS GnlModifyComponentIntoLogicFunction (Gnl, UserCompo, GnlLib)
    GNL                               Gnl;
    GNL_USER_COMPONENT                UserCompo;
    LIBC_LIB                           GnlLib;
{
    int                                i;
    LIBC_CELL                         Cell;
    LIBC_PIN                           Pins;
    BLIST                             ListOutputPin;
    LIBC_PIN                           OutputPin;
    LIBC_BOOL_OPR                     Function;
    BLIST                             Interface;
    GNL_NODE                          Node;
}

```

gnlverif.c

```
char *pinName;
GNL_VAR OutVar;
GNL_FUNCTION NewFunction;
LIBC_NAME_LIST ListPinName;
char *UserCompoName;

Interface = GnlUserComponentInterface (UserCompo);
UserCompoName = GnlUserComponentName (UserCompo);

Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompo);
Pins = LibCellPins (Cell);

/* we extract the first output Pin. If no output pin then we return */
if (GnlGetListOutputPinFromPins (Pins, &ListOutputPin))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListOutputPin); i++)
{
    OutputPin = (LIBC_PIN)BListElt (ListOutputPin, i);

    if (!OutputPin)
    {
        fprintf (stderr, " WARNING: Cell <%s> has no output pin\n",
                 LibCellName (Cell));

        return (GNL_OK);
    }

    ListPinName = LibPinName (OutputPin);

    /* Since 'PinName' can be a bundle a bus or a pin group we need to
*/
    /* that it is a single bit signal.
*/
    /* if there are different names we do not take this cell into
account*/
    if (!GnlSinglePinName (ListPinName))
    {
        fprintf (stderr,
                 " WARNING: Cell <%s> has a complex output pin name\n",
                 LibCellName (Cell));
        exit (1);
    }

    PinName = LibNameListName (ListPinName);
    Function = LibPinFunction (OutputPin);

    if (GnlGetGnlNodeFromBoolOprWithInterface (Gnl, UserCompoName,
                                                Interface, Function, &Node))
        return (GNL_MEMORY_FULL);

    if (!GnlGetActualVarOfFormalName (Interface, PinName, &OutVar))
    {
        fprintf (stderr,
                 " ERROR: cannot find output pin <%s> in Cell <%s>\n",
                 PinName, CellName (Cell));
    }
}
```

```

        PinName,
        LibCellName (Cell));
    exit (1);
}

if (GnlVarFunction (OutVar))
{
    fprintf (stderr,
            " ERROR: Signal <%s> is a multi-source signal\n",
            GnlVarName (OutVar));
    exit (1);
}

if (GnlFunctionCreate (Gnl, OutVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (OutVar, NewFunction);
SetGnlFunctionOnSet (NewFunction, Node);

if (BListAddElt (GnlFunctions (Gnl), (int)OutVar))
    return (GNL_MEMORY_FULL);

}

GnlFreeUserComponent (UserCompo);

BListQuickDelete (&ListOutputPin);

return (GNL_OK);
}

/*
 * GnlModifyComponentIntoLogicFunctionForNLDelay
 */
GNL_STATUS GnlModifyComponentIntoLogicFunctionForNLDelay (Gnl, UserCompo,
                                                       GnlLib)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        UserCompo;
    LIBC_LIB                  GnlLib;
    {
        int                     i;
        LIBC_CELL                Cell;
        LIBC_PIN                 Pins;
        LIBC_PIN                 OutputPin;
        BLIST                    ListOutputPin;
        LIBC_BOOL_OPR             Function;
        BLIST                    Interface;
        GNL_NODE                 Node;
        char                     *PinName;
        GNL_VAR                  OutVar;
        GNL_FUNCTION              NewFunction;
        LIBC_NAME_LIST            ListPinName;
        char                     *UserCompoName;
        GNL_MAP_NODE_INFO         NewMapNodeInfo;
    }
}

```

gnlverif.c

```
Interface = GnlUserComponentInterface (UserCompo);
UserCompoName = GnlUserComponentName (UserCompo);

Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompo);
Pins = LibCellPins (Cell);

/* we extract the first output Pin. If no output pin then we return */
if (GnlGetListOutputPinFromPins (Pins, &ListOutputPin))
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListOutputPin); i++)
{
    OutputPin = (LIBC_PIN)BListElt (ListOutputPin, i);

    if (!OutputPin)
    {
        fprintf (stderr, " WARNING: Cell <%s> has no output pin\n",
                LibCellName (Cell));

        return (GNL_OK);
    }

    ListPinName = LibPinName (OutputPin);

    /* Since 'PinName' can be a bundle a bus or a pin group we need to
     */
    /* that it is a single bit signal.
     */
    /* if there are different names we do not take this cell into
account*/
    if (!GnlSinglePinName (ListPinName))
    {
        fprintf (stderr,
                 " WARNING: Cell <%s> has a complex output pin name\n",
                 LibCellName (Cell));
        exit (1);
    }

    PinName = LibNameListName (ListPinName);
    Function = LibPinFunction (OutputPin);

    if (GnlGetGnlNodeFromBoolOprWithInterface (Gnl, UserCompoName,
                                                Interface, Function, &Node))
        return (GNL_MEMORY_FULL);

    if (!GnlGetActualVarOfFormalName (Interface, PinName, &OutVar))
    {
        fprintf (stderr,
                 " ERROR: cannot find output pin <%s> in Cell <%s>\n",
                 PinName,
                 LibCellName (Cell));
        exit (1);
    }

    if (GnlVarFunction (OutVar))
    {
```

gnlverif.c

```
fprintf (stderr,
        " ERROR: Signal <%s> is a multi-source signal\n",
        GnlVarName (OutVar));
    exit (1);
}

/* we add extra info on this node for the futur NL delay mapping*/
if (GnlMapNodeInfoCreate (&NewMapNodeInfo))
    return (GNL_MEMORY_FULL);
SetGnlNodeHook (Node, NewMapNodeInfo);

/* We store the original Compo for this node corresponding to      */
/* the original combinational gate.                                */
SetGnlMapNodeInfoOriginalCompo (Node, UserCompo);

/* toto */
/*
 * SetGnlMapNodeInfoOriginalCompo (Node, NULL);
 */

if (GnlFunctionCreate (Gnl, OutVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (OutVar, NewFunction);
SetGnlFunctionOnSet (NewFunction, Node);

if (BListAddElt (GnlFunctions (Gnl), (int)OutVar))
    return (GNL_MEMORY_FULL);
}

BListQuickDelete (&ListOutputPin);

return (GNL_OK);
}

/*
 * GnlModifyGnlComponentsWithCells
 */
/*
 * This procedure replaces the GNL_USER_COMPONENT according to their
 * associated libc cell. They can be replaced by
 * GNL_SEQUENTIAL_COMPONENT if the associated linking cell is a Dff, ...
 * If 'ReplaceCombi' is 1 then the combinatorial cells are replaced
 * by the Boolean function they represent, otherwise they are not
 * replaced.
 */
/*
 * GNL_STATUS GnlModifyGnlComponentsWithCells (Gnl, GnlLib, ReplaceCombi,
 *                                             GetEquivDeriveCells)
 */
{
    GNL          Gnl;
    LIBC_LIB     GnlLib;
    int          ReplaceCombi;
    int          GetEquivDeriveCells;
    {
        BLIST          Components;
        int            i;
        GNL_COMPONENT   ComponentI;
        GNL_USER_COMPONENT UserCompoI;
        LIBC_CELL       Cell;
        GNL_SEQUENTIAL_COMPONENT SeqCompoI;
```

gnlverif.c

```
LIBC_PIN           OutPin;
GNL_TRISTATE_COMPONENT TristateCompoI;
GNL_TRISTATE_COMPO_INFO NewTriStateCompoInfo;
GNL_VAR            OutVarBar;
GNL_NODE           NewNode;
GNL_NODE           NodeNot;
GNL_VAR            NewVar;
GNL_FUNCTION       NewFunction;
LIB_CELL           MotherCell;
BLIST              ListDeriveCells;
BDD                Bdd;
BDD_PTR            BddPtr;
LIB_DERIVE_CELL   FirstDeriveCell;
int                ReplaceBuffer;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT) BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT) ComponentI;

    /* There is a Gnl associated so it is not linked to a cell */
    if (GnlUserComponentGnlDef (UserCompoI))
        continue;

    /* If no linked libc cell then is it a black box */
    if (!GnlUserComponentCellDef (UserCompoI))
        continue;

    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompoI);
    if (LibCellDontUse (Cell))
    {
        fprintf (stderr,
                  " WARNING: Netlist using DONT USE cell [%s]\n",
                  LibCellName (Cell));
    }

    /* Case where the component is linked to a sequential cell */
    if (LibCellFFLatch (Cell))
    {
        if (GnlModifyComponentIntoSeqComponent (Gnl, UserCompoI,
                                                GnlLib, ReplaceCombi, &SeqCompoI))
            return (GNL_MEMORY_FULL);
        BListElt (Components, i) = (int) SeqCompoI;

        if (ReplaceCombi)
        {
            if (!GnlSequentialCompoOutput (SeqCompoI))
            {
                OutVarBar = GnlSequentialCompoOutputBar (SeqCompoI);
                if (GnlCreateUniqueVar (Gnl,
                                       GnlVarName (OutVarBar), &NewVar))

```

0 9 8 7 6 5 4 3 2 1 0
* 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

        return (GNL_MEMORY_FULL);

    if (GnlCreateNodeForVar (Gnl, NewVar, &NewNode))
        return (GNL_MEMORY_FULL);

    if (GnlCreateNodeNot (Gnl, NewNode, &NodeNot))
        return (GNL_MEMORY_FULL);

    if (GnlFunctionCreate (Gnl, OutVarBar, NULL,
                           &NewFunction))
        return (GNL_MEMORY_FULL);

    if (GnlVarFunction (OutVarBar))
    {
        fprintf (stderr,
        " ERROR: Signal <%s> is a multi-source signal\n",
        GnlVarName (OutVarBar));
        exit (1);
    }
    SetGnlVarFunction (OutVarBar, NewFunction);
    SetGnlFunctionOnSet (NewFunction, NodeNot);

    SetGnlSequentialCompoOutput (SeqCompoI, NewVar);
    SetGnlVarDir (NewVar, GNL_VAR_LOCAL_WIRING);

    SetGnlSequentialCompoOutputBar (SeqCompoI, NULL);

    if (BListAddElt (GnlFunctions (Gnl), (int)OutVarBar))
        return (GNL_MEMORY_FULL);
    }

}

continue;
}

/* Case where the component is linked to a 3 states */
if (LibCellWith3StatePin (Cell, &OutPin))
{
    if (!Gnl3StateCellCanBeSupported (Cell))
    {
        fprintf (stderr,
        " ERROR: the libc cell [%s] cannot be processed by %s\n",
        LibCellName (Cell), GNL_TOOL_NAME);
        exit (1);
    }

    if (GnlModifyComponentIntoTristateComponent (Gnl, UserCompoI,
                                                GnlLib, ReplaceCombi, &TristateCompoI))
        return (GNL_MEMORY_FULL);
    BListElt (Components, i) = (int)TristateCompoI;

    continue;
}

/* The user asked to remove the original buffers.           */
ReplaceBuffer = 0;

```

gnlverif.c

```
if (GnlEnvRemoveBuffer ())
{
    if (GnlCellIsBuffer (Cell))
    {
        ReplaceBuffer = 1;
        fprintf (stderr,
            " WARNING: Removing original buffer '%s' '%s'\n",
            GnlUserComponentName (UserCompoI),
            GnlUserComponentInstName (UserCompoI));
    }
}

/* Otherwise it is a combinatorial Cell */
```

```
/* We do not replace combinatorial components if we say so */
if (!ReplaceCombi && !ReplaceBuffer)
    continue;
```

```
if (GetEquivDeriveCells)
{
    /* We extract the list of derive equivalent cells of the */
    /* 'cell'. */
    MotherCell = (LIB_CELL)LibCellHook (Cell);
    ListDeriveCells = LibHCellDeriveCells (MotherCell);

    if (ListDeriveCells)
    {
        FirstDeriveCell = (LIB_DERIVE_CELL)BListElt (ListDeriveCells,
            0);
        Bdd = LibDeriveCellBdd (FirstDeriveCell);
        BddPtr = GetBddPtrFromBdd (Bdd);
        ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
    }
    SetGnlUserComponentEquivCells (UserCompoI, ListDeriveCells);
}
```

```
/* We replace the combinatorial Cell by its boolean function. */
if (GnlModifyComponentIntoLogicFunction (Gnl, UserCompoI,
    GnlLib))
    return (GNL_MEMORY_FULL);

/* we remove the User component corresponding to the cell */
/* because we replaced it by a Function GNL_NODE */
BListDelInsert (Components, i+1);
i--;
}
```

```
return (GNL_OK);
}

/*-----*/
/* GnlModifyGnlComponentsWithCombinatorialCells */
/*-----*/
/* This procedure replaces the GNL_USER_COMPONENT according to their */
/* associated libc cell. Only combinatorial elements will be replaced. */
/*-----*/
```

gnlverif.c

```
GNL_STATUS GnlModifyGnlComponentsWithCombinatorialCells (Gnl, GnlLib)
{
    GNL          Gnl;
    LIBC_LIB     GnlLib;
{
    BLIST          Components;
    int           i;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    LIBC_CELL      Cell;
    GNL_SEQUENTIAL_COMPONENT SeqCompoI;
    LIBC_PIN       OutPin;
    GNL_TRISTATE_COMPONENT TristateCompoI;
    GNL_TRISTATE_COMPO_INFO NewTriStateCompoInfo;
    GNL_VAR        OutVarBar;
    GNL_NODE       NewNode;
    GNL_NODE       NodeNot;
    GNL_VAR        NewVar;
    GNL_FUNCTION   NewFunction;
    LIB_CELL       MotherCell;
    BLIST          ListDeriveCells;
    BDD            Bdd;
    BDD_PTR        BddPtr;
    LIB_DERIVE_CELL FirstDeriveCell;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;

    /* There is a Gnl associated so it is not linked to a cell */
    if (GnlUserComponentGnlDef (UserCompoI))
        continue;

    /* If no linked libc cell then is it a black box */
    if (!GnlUserComponentCellDef (UserCompoI))
        continue;

    Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompoI);
    if (LibCellDontUse (Cell))
    {
        fprintf (stderr,
                  " WARNING: Netlist using DONT USE cell [%s]\n",
                  LibCellName (Cell));
    }

    /* Case where the component is linked to a sequential cell */
    if (LibCellFFLatch (Cell))
        continue;

    /* Case where the component is linked to a 3 states */
    if (LibCellWith3StatePin (Cell, &OutPin))
```

```

        continue;

    /* Case where the component is linked to a buffer.          */
    if (GnlCellIsBuffer (Cell))
        continue;

    /* Otherwise it is a combinatorial Cell                      */

    /* We extract the list of derive equivalent cells of the 'cell'.*/
    MotherCell = (LIB_CELL)LibCellHook (Cell);
    ListDeriveCells = LibHCellDeriveCells (MotherCell);

    if (ListDeriveCells)
    {
        FirstDeriveCell = (LIB_DERIVE_CELL)BListElt (ListDeriveCells,
                                                       0);
        Bdd = LibDeriveCellBdd (FirstDeriveCell);
        BddPtr = GetBddPtrFromBdd (Bdd);
        ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
    }
    SetGnlUserComponentEquivCells (UserCompoI, ListDeriveCells);

    /* We replace the combinatorial Cell by its boolean function. */
    if (GnlModifyComponentIntoLogicFunction (Gnl, UserCompoI,
                                              GnlLib))
        return (GNL_MEMORY_FULL);

    /* we remove the User component corresponding to the cell */
    /* because we replaced it by a Function GNL_NODE           */
    BListDelInsert (Components, i+1);
    i--;
}

return (GNL_OK);
}

/*-----
/* GnlReplaceUserCompoByBufferForNLDelayMapping             */
/*-----*/
/* This procedure replaces a user component by a function node of the */
/* form: y = x; where 'x' is the input of the buffer, 'y' its output. */
/* Moreover, the node representing 'x' pointed by 'y' has a reference to */
/* the original buffer (thru field 'GnlMapNodeInfoOriginalCell(node)'). */
/*-----*/
GNL_STATUS GnlReplaceUserCompoByBufferForNLDelayMapping (Gnl, UserCompo,
                                                       Cell)
{
    GNL                      Gnl;
    GNL_USER_COMPONENT        UserCompo;
    LIBC_CELL                 Cell;
{
    BLIST                    Interface;
    char                     *UserCompoName;
    int                      i;
    LIBC_PIN                 PinIn;
    LIBC_PIN                 PinOut;
    LIBC_PIN                 Pins;
}

```

gnlverif.c

```
LIBC_NAME_LIST ListPinName;
char           *PinInName;
char           *PinOutName;
GNL_VAR        Input;
GNL_VAR        Output;
GNL_FUNCTION   NewFunction;
GNL_NODE       NewNode;
GNL_MAP_NODE_INFO NewMapNodeInfo;

Interface = GnlUserComponentInterface (UserCompo);
UserCompoName = GnlUserComponentName (UserCompo);

i = 0;
PinIn = PinOut = NULL;
Pins = LibCellPins (Cell);
for (; Pins != NULL; Pins = LibPinNext(Pins))
{
    if (LibPinDirection (Pins) == OUTPUT_E)
        PinOut = Pins;
    if (LibPinDirection (Pins) == INPUT_E)
        PinIn = Pins;
    i++;
}

if (i != 2)
{
    fprintf (stderr,
    "\n ERROR: cannot handle buffer with more than two pins: '%s'\n",
    LibCellName (Cell));
    exit (1);
}

if ((PinIn == NULL) || (PinOut == NULL))
{
    fprintf (stderr,
    "\n ERROR: cannot find all pins of buffer: '%s'\n",
    LibCellName (Cell));
    exit (1);
}

ListPinName = LibPinName (PinIn);
if (!GnlSinglePinName (ListPinName))
{
    fprintf (stderr,
    " WARNING: Cell <%s> has a complex input pin name\n",
    LibCellName (Cell));
    exit (1);
}
PinInName = LibNameListName (ListPinName);

ListPinName = LibPinName (PinOut);
if (!GnlSinglePinName (ListPinName))
{
    fprintf (stderr,
    " WARNING: Cell <%s> has a complex output pin name\n",
    LibCellName (Cell));
```

gnlverif.c

```
    exit (1);
}
PinOutName = LibNameListName (ListPinName);

if (!GnlGetActualVarOfFormalName (Interface, PinInName, &Input))
{
    fprintf (stderr,
" ERROR: cannot find actual parameter associated to Input pin <%s> in
<%s>\n",
                PinInName, GnlUserComponentInstName (UserCompo));
    exit (1);
}

if (!GnlGetActualVarOfFormalName (Interface, PinOutName, &Output))
{
    fprintf (stderr,
" ERROR: cannot find actual parameter associated to Output pin <%s> in
<%s>\n",
                PinOutName, GnlUserComponentInstName (UserCompo));
    exit (1);
}

if (GnlFunctionCreate (Gnl, Output, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);
SetGnlVarFunction (Output, NewFunction);

if (GnlCreateNodeForVar (Gnl, Input, &NewNode))
    return (GNL_MEMORY_FULL);

/* we add extra info on this node for the futur NL delay mapping */
if (GnlMapNodeInfoCreate (&NewMapNodeInfo))
    return (GNL_MEMORY_FULL);
SetGnlNodeHook (NewNode, NewMapNodeInfo);

/* We store the original Compo for this node which is a buffer. */
SetGnlMapNodeInfoOriginalCompo (NewNode, UserCompo);

/* toto */
/*
    SetGnlMapNodeInfoOriginalCompo (NewNode, NULL);
*/

SetGnlFunctionOnSet (NewFunction, NewNode);
if (BListAddElt (GnlFunctions (Gnl), (int)Output))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * GnlModifyGnlComponentsWithCombinatorialCellsForNLdelayMapping
 */
/*
 * This procedure replaces the GNL_USER_COMPONENT according to their
 * associated libc cell. Only combinatorial elements will be replaced.
 */
/*
 * GNL_STATUS GnlModifyGnlComponentsWithCombinatorialCellsForNLdelayMapping

```

```

(Gnl, GnlLib)

GNL          Gnl;
LIBC_LIB     GnlLib;
{
    BLIST          Components;
    int            i;
    GNL_COMPONENT  ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    LIBC_CELL      Cell;
    GNL_SEQUENTIAL_COMPONENT SeqCompoI;
    LIBC_PIN       OutPin;
    GNL_TRISTATE_COMPONENT TristateCompoI;
    GNL_TRISTATE_COMPO_INFO  NewTriStateCompoInfo;
    GNL_VAR        OutVarBar;
    GNL_NODE       NewNode;
    GNL_NODE       NodeNot;
    GNL_VAR        NewVar;
    GNL_FUNCTION   NewFunction;
    LIB_CELL       MotherCell;
    BLIST          ListDeriveCells;
    BDD            Bdd;
    BDD_PTR        BddPtr;
    LIB_DERIVE_CELL FirstDeriveCell;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;

    /* There is a Gnl associated so it is not linked to a cell */
    if (GnlUserComponentGnlDef (UserCompoI))
        continue;

    /* If no linked libc cell then is it a black box */
    if (!GnlUserComponentCellDef (UserCompoI))
        continue;

    Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompoI);
    if (LibCellDontUse (Cell))
    {
        fprintf (stderr,
                 " WARNING: Netlist using DONT USE cell [%s]\n",
                 LibCellName(Cell));
    }

    /* Case where the component is linked to a sequential cell */
    if (LibCellFFLatch (Cell))
        continue;

    /* Case where the component is linked to a 3 states */
    if (LibCellWith3StatePin (Cell, &OutPin))

```

```

        continue;

/* Case where the component is linked to a buffer.          */
if (GnlCellIsBuffer (Cell))
{
    if (GnlReplaceUserCompoByBufferForNLDelayMapping (Gnl,
        UserCompoI, Cell))
        return (GNL_MEMORY_FULL);

    BListDelInsert (Components, i+1);
    i--;
    continue;
}

/* Otherwise it is a combinatorial Cell                      */

/* We extract the list of derive equivalent cells of the 'cell'.*/
MotherCell = (LIB_CELL)LibCellHook (Cell);
ListDeriveCells = LibHCellDeriveCells (MotherCell);

if (ListDeriveCells)
{
    FirstDeriveCell = (LIB_DERIVE_CELL)BListElt (ListDeriveCells,
        0);
    Bdd = LibDeriveCellBdd (FirstDeriveCell);
    BddPtr = GetBddPtrFromBdd (Bdd);
    ListDeriveCells = LibBddInfoDeriveCells (BddPtr);
}
SetGnlUserComponentEquivCells (UserCompoI, ListDeriveCells);

/* We replace the combinatorial Cell by its boolean function. */
if (GnlModifyComponentIntoLogicFunctionForNLDelay (Gnl,
        UserCompoI, GnlLib))
    return (GNL_MEMORY_FULL);

/* we remove the User component corresponding to the cell */
/* because we replaced it by a Function GNL_NODE           */
BListDelInsert (Components, i+1);
i--;
}

return (GNL_OK);
}

/*
*-----*
/* GnlReplaceAllComponentsWithLinkLibCellsRec               */
/*-----*
/* This procedure replaces recursively each Gnl of the network 'Nw' all */
/* the GNL_USER_COMPONENT according to their associated libc cell. They */
/* can be replaced by GNL_SEQUENTIAL_COMPONENT if the associated linking*/
/* cell is a Dff, etc ...                                     */
/* If 'ReplaceCombi' is 1 then the combinatorial cells are replaced   */
/* by the Boolean function they represent, otherwise they are not   */
/* replaced.                                                 */
/*-----*
GNL_STATUS GnlReplaceAllComponentsWithLinkLibCellsRec (Nw, Gnl, GnlLib,

```

gnlverif.c

```
ReplaceCombi, GetEquivDeriveCells)

GNL_NETWORK Nw;
GNL Gnl;
LIBC_LIB GnlLib;
int ReplaceCombi;
int GetEquivDeriveCells;
{
    int i;
    BLIST Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL GnlCompoI;

    /* Already processed. */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw)) /*/
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));

    if (GnlModifyGnlComponentsWithCells (Gnl, GnlLib, ReplaceCombi,
                                        GetEquivDeriveCells))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT) ComponentI;
        if (!GnlUserComponentGnlDef (UserCompoI))
            continue;

        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

        if (GnlReplaceAllComponentsWithLinkLibCellsRec (Nw, GnlCompoI,
                                                       GnlLib, ReplaceCombi, GetEquivDeriveCells))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----
/* GnlReplaceOnlyCombinatorialComponentsWithLinkLibCellsRec */ */
/*-----*/
/* This procedure replaces recursively in each Gnl of the network 'Nw' */
/* the GNL_USER_COMPONENT according to their associated libc cell. */
/* Only combinatorial components will be replaced. */
/*-----*/
GNL_STATUS GnlReplaceOnlyCombinatorialComponentsWithLinkLibCellsRec (Nw,
                                                                    Gnl, GnlLib)
GNL_NETWORK Nw;
GNL Gnl;
```

gnlverif.c

```
LIBC_LIB      GnlLib;
{
    int          i;
    BLIST        Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL          GnlCompoI;

    /* Already processed. */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))           */
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));

    if (GnlModifyGnlComponentsWithCombinatorialCells (Gnl, GnlLib))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT)ComponentI;
        if (!GnlUserComponentGnlDef (UserCompoI))
            continue;

        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

        if (GnlReplaceOnlyCombinatorialComponentsWithLinkLibCellsRec (Nw,
            GnlCompoI, GnlLib))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 * GnlReplaceComponentsWithLinkLibCellsForNLDelayMappingRec
 */
/*
 * This procedure replaces recursively in each Gnl of the network 'Nw'   */
/* the GNL_USER_COMPONENT according to their associated libc cell.   */
/* Only combinatorial components will be replaced.                   */
/*
 * GNL_STATUS GnlReplaceComponentsWithLinkLibCellsForNLDelayMappingRec (Nw,
 * Gnl, GnlLib)
 */
GNL_NETWORK Nw;
GNL          Gnl;
LIBC_LIB     GnlLib;
{
    int          i;
    BLIST        Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
```

gnlverif.c

```
GNL GnlCompoI;

/* Already processed. */
if (GnlTag (Gnl) == GnlNetworkTag (Nw))
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

if (GnlModifyGnlComponentsWithCombinatorialCellsForNLdelayMapping (
    Gnl, GnlLib))
    return (GNL_MEMORY_FULL);

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    if (!GnlUserComponentGnlDef (UserCompoI))
        continue;

    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (GnlReplaceComponentsWithLinkLibCellsForNLDelayMappingRec (Nw,
        GnlCompoI, GnlLib))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlResetLeafDAGNodeInputs
 */
void GnlResetLeafDAGNodeInputs (Node)
    GNL_NODE Node;
{
    int i;
    GNL_NODE NodeI;
    GNL_VAR Var;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        Var = (GNL_VAR)GnlNodeSons (Node);
        SetGnlVarHook (Var, NULL);
        return;
    }

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
```

gnlverif.c

```
{  
    NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);  
    GnlResetLeafDAGNodeInputs (NodeI);  
}  
}  
  
/*-----*/  
/* GnlLeafDAGNodeRec */  
/*-----*/  
int GnlLeafDAGNodeRec (Node)  
    GNL_NODE Node;  
{  
    int          i;  
    int          MaxFanOut;  
    int          MaxFanOutI;  
    GNL_NODE NodeI;  
    GNL_VAR Var;  
  
    if (GnlNodeOp (Node) == GNL_CONSTANTE)  
        return (0);  
  
    if (GnlNodeOp (Node) == GNL_VARIABLE)  
    {  
        Var = (GNL_VAR)GnlNodeSons (Node);  
        if (GnlVarDir (Var) == GNL_VAR_INPUT)  
            return (1);  
  
        SetGnlVarHook (Var, (int)GnlVarHook (Var)+1);  
        return ((int)GnlVarHook (Var));  
    }  
  
    MaxFanOut = 0;  
  
    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)  
    {  
        NodeI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);  
        MaxFanOutI = GnlLeafDAGNodeRec (NodeI);  
        if (MaxFanOutI > MaxFanOut)  
            MaxFanOut = MaxFanOutI;  
    }  
  
    return (MaxFanOut);  
}  
  
/*-----*/  
/* GnlLeafDAGNode */  
/*-----*/  
int GnlLeafDAGNode (Node)  
    GNL_NODE Node;  
{  
    int          MaxFanout;  
  
    GnlResetLeafDAGNodeInputs (Node);  
  
    MaxFanout = GnlLeafDAGNodeRec (Node);
```

```

        return ((MaxFanout > 1));
    }

/*-----
/* GnlRemoveCompoNotToKeepOnNode
/*-----
/* removes the component association of all the nodes having all inputs */
/* uni-fanout. The others are kept.
/*-----*/
GNL_STATUS GnlRemoveCompoNotToKeepOnNode (Gnl, Node, Keep, NotKeep)
{
    GNL          Gnl;
    GNL_NODE     Node;
    int          *Keep;
    int          *NotKeep;
{
    int          i;
    GNL_USER_COMPONENT UserCompo;
    BLIST        Interface;
    GNL_ASSOC     AssocI;
    GNL_VAR      Actual;
    GNL_NODE     SonI;
    int          KeepCompo;
    char         *Formal;
    LIBC_PIN     PinFormal;
    LIBC_CELL    Cell;
    int          NKeep;
    int          NNotKeep;

    *Keep = *NotKeep = 0;

    if (GnlNodeOp (Node) == GNL_VARIABLE)
    {
        /* case of buffer.
        if (GnlNodeHook (Node) && (GnlMapNodeInfoOriginalCompo (Node)))
            *Keep = 1;

        return (GNL_OK);
    }

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    /* If the node has an associated compo. */
    if (GnlNodeHook (Node) && (GnlMapNodeInfoOriginalCompo (Node)))
    {
        KeepCompo = 0;
        UserCompo = GnlMapNodeInfoOriginalCompo (Node);
        Cell = GnlUserComponentCellDef (UserCompo);

        Interface = GnlUserComponentInterface (UserCompo);
        for (i=0; i<BListSize (Interface); i++)
        {
            AssocI = (GNL_ASSOC)BListElt (Interface, i);
            if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
            {

```

```

Formal = (char*)GnlAssocFormalPort (AssocI);
PinFormal = GnlGetPinCellWithName (Cell, Formal);

if (!PinFormal)
{
    fprintf (stderr,
        " ERROR: cannot find pin name <%s> in cell [%s]\n",
        Formal, LibCellName (Cell));
    exit (1);
}

if (LibPinDirection (PinFormal) == OUTPUT_E)
    continue;
}

Actual = GnlAssocActualPort (AssocI);
if (!GnlVarIsVar (Actual))
{
    fprintf (stderr,
        "\n ERROR: cannot handle actual vector here\n");
    exit (1);
}

/* The component has an input with multi-fanout which is      */
/* the ouput of another comb. cell.                           */
if ((GnlVarHook (Actual) > (void*)1) &&
    (GnlVarDir (Actual) != GNL_VAR_INPUT) &&
    (GnlVarDir (Actual) != GNL_VAR_LOCAL_WIRING))
{
    KeepCompo = 1;
    break;
}
}

if (!KeepCompo)
{
    /* Actually the node can correspond to a leaf-DAG cell (xor) */
    /* so we recompute the fanout of the inputs because some of */
    /* them can be really multi-fanout inputs.                  */
    if (GnlLeafDAGNode (Node))
    {
        (*Keep)++;
    }
    else
    {
        /* we remove the component association.                   */
        fprintf (stderr, " Remove association 'Ins=%s Comp=%s'\n",
            GnlUserComponentInstName (UserCompo),
            GnlUserComponentName (UserCompo));
        SetGnlMapNodeInfoOriginalCompo (Node, NULL);

        (*NotKeep)++;
    }
}
else
{
    (*Keep)++;
}

```

gnlverif.c

```
    }

}

for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
{
    SonI = (GNL_NODE) BListElt (GnlNodeSons (Node), i);
    if (GnlRemoveCompoNotToKeepOnNode (Gnl, SonI, &NKeep, &NNotKeep))
        return (GNL_MEMORY_FULL);

    *Keep += NKeep;
    *NotKeep += NNotKeep;
}

return (GNL_OK);
}

/*-----
/* GnlComputeCapaFromUserCompo
/*-----*/
void GnlComputeCapaFromUserCompo (Gnl, GnlLibc, UserCompo, Cell)
    GNL                      Gnl;
    LIBC_LIB                  GnlLibc;
    GNL_USER_COMPONENT         UserCompo;
    LIBC_CELL                  Cell;
{
    int                         i;
    BLIST                       Interface;
    GNL_ASSOC                   AssocI;
    GNL_VAR                     Actual;
    LIBC_PIN                    PinFormal;
    float                       PinCapa;
    char                        *Formal;

    Interface = GnlUserComponentInterface (UserCompo);
    for (i=0; i<BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        PinFormal = NULL;
        if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
        {
            Formal = (char*) GnlAssocFormalPort (AssocI);
            PinFormal = GnlGetPinCellWithName (Cell, Formal);

            if (!PinFormal)
            {
                fprintf (stderr,
                    " ERROR: cannot find pin name <%s> in cell [%s]\n",
                    Formal, LibCellName (Cell));
                exit (1);
            }

            if (LibPinDirection (PinFormal) == OUTPUT_E)
                continue;
        }
    }
}
```

gnlverif.c

```
if (!PinFormal)
{
    fprintf (stderr,
             "\n ERROR: cannot find some formal pin\n");
    exit (1);
}

Actual = GnlAssocActualPort (AssocI);
if (!GnlVarIsVar (Actual))
{
    fprintf (stderr,
             "\n ERROR: cannot handle actual vector here\n");
    exit (1);
}

PinCapa = GnlGetCapaFromPin (PinFormal);

/* Cumulating the capacitance at var 'Actual'. */
SetGnlVarOutCapa (Actual, GnlVarOutCapa (Actual) + PinCapa);
}

/*
 * GnlComputeCapaFromKeptCompoOnNode
 */
GNL_STATUS GnlComputeCapaFromKeptCompoOnNode (Gnl, GnlLibc, Node)
    GNL          Gnl;
    LIBC_LIB GnlLibc;
    GNL_NODE Node;
{
    int           i;
    GNL_USER_COMPONENT UserCompo;
    GNL_NODE      SonI;
    LIBC_CELL     Cell;

    if (GnlNodeOp (Node) == GNL_CONSTANTE)
        return (GNL_OK);

    /* If the node has an associated compo. */
    if (GnlNodeHook (Node) && (GnlMapNodeInfoOriginalCompo (Node)))
    {
        UserCompo = GnlMapNodeInfoOriginalCompo (Node);
        Cell = GnlUserComponentCellDef (UserCompo);

        GnlComputeCapaFromUserCompo (Gnl, GnlLibc, UserCompo, Cell);
    }

    if (GnlNodeOp (Node) == GNL_VARIABLE)
        return (GNL_OK);

    for (i=0; i<BListSize (GnlNodeSons (Node)); i++)
    {
        SonI = (GNL_NODE)BListElt (GnlNodeSons (Node), i);
        if (GnlComputeCapaFromKeptCompoOnNode (Gnl, GnlLibc, SonI))
            return (GNL_MEMORY_FULL);
    }
}
```

```

        return (GNL_OK);
    }

/*
 * GnlRemoveCompoNotToKeep
 */
GNL_STATUS GnlRemoveCompoNotToKeep (Gnl, GnlLib, Keep, NotKeep)
    GNL          Gnl;
    LIBC_LIB     GnlLib;
    int          *Keep;
    int          *NotKeep;
{
    int           i;
    GNL_VAR      VarI;
    GNL_FUNCTION Function;
    int          NKeep;
    int          NNotKeep;

    *Keep = *NotKeep = 0;

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        if (!GnlVarFunction (VarI))
            continue;

        Function = GnlVarFunction (VarI);

        if (GnlRemoveCompoNotToKeepOnNode (Gnl,
                                           GnlFunctionOnSet (Function),
                                           &NKeep, &NNotKeep))
            return (GNL_MEMORY_FULL);

        *Keep += NKeep;
        *NotKeep += NNotKeep;
    }

    return (GNL_OK);
}

/*
 * GnlRemoveCompoNotToKeepRec
 */
GNL_STATUS GnlRemoveCompoNotToKeepRec (Nw, Gnl, GnlLib)
    GNL_NETWORK  Nw;
    GNL          Gnl;
    LIBC_LIB     GnlLib;
{
    int           i;
    BLIST         Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL          GnlCompoI;
    int          Keep;
    int          NotKeep;
}

```

```

/* Already processed. */ 
if (GnlTag (Gnl) == GnlNetworkTag (Nw))
    return (GNL_OK);

SetGnlTag (Gnl, GnlNetworkTag (Nw));

if (GnlRemoveCompoNotToKeep (Gnl, GnlLib, &Keep, &NotKeep))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " GNL = '%s', KEEP = %d, NOT_KEEP = %d\n",
        GnlName (Gnl), Keep, NotKeep);

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;

    UserCompoI = (GNL_USER_COMPONENT)ComponentI;
    if (!GnlUserComponentGnlDef (UserCompoI))
        continue;

    GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (GnlRemoveCompoNotToKeepRec (Nw, GnlCompoI, GnlLib))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlGetNLDelayArea
 */
GNL_STATUS GnlGetNLDelayArea (Nw, Gnl, GnlLib)
    GNL_NETWORK      Nw;
    GNL             Gnl;
    LIBC_LIB        GnlLib;
{
    BLIST          Components;
    int            i;
    GNL_COMPONENT  ComponentI;
    GNL_USER_COMPONENT  UserCompoI;
    LIBC_CELL      Cell;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);

    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
}

```

```

UserCompoI = (GNL_USER_COMPONENT)ComponentI;

/* There is a Gnl associated so it is not linked to a cell      */
if (GnlUserComponentGnlDef (UserCompoI))
    continue;

/* If no linked libc cell then is it a black box                  */
if (!GnlUserComponentCellDef (UserCompoI))
    continue;

Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompoI);

SetGnlNetworkNLDelayArea (Nw, GnlNetworkNLDelayArea (Nw) +
                           LibCellArea (Cell));
}

return (GNL_OK);
}

/*-----*/
/* GnlGetNLDelayAreaRec                                         */
/*-----*/
GNL_STATUS GnlGetNLDelayAreaRec (Nw, Gnl, GnlLib)
{
    GNL_NETWORK    Nw;
    GNL           Gnl;
    LIBC_LIB       GnlLib;
    int            i;
    BLIST          Components;
    GNL_COMPONENT   ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL           GnlCompoI;

    if (GnlGetNLDelayArea (Nw, Gnl, GnlLib))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT)ComponentI;
        if (!GnlUserComponentGnlDef (UserCompoI))
            continue;

        GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

        if (GnlGetNLDelayAreaRec (Nw, GnlCompoI, GnlLib))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

```

```

}

/*-----*/
/* GnlComputeCapacitancesFromKeptCompo */
/*-----*/
GNL_STATUS GnlComputeCapacitancesFromKeptCompo (Nw, Gnl, GnlLib)
    GNL_NETWORK      Nw;
    GNL           Gnl;
    LIBC_LIB GnlLib;
{
    int                  i;
    GNL_VAR             VarI;
    GNL_FUNCTION        Function;
    LIBC_WIRE_LOAD_SELECT WireLoadSelect;
    BLIST               Components;
    GNL_COMPONENT       ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    LIBC_CELL           Cell;
    LIBC_PIN            OutPin;
    int                 j;
    BLIST               BucketI;
    GNL_VAR             VarJ;
    int                 Fanout;
    float               Length;
    float               WireCapa;
    BLIST               HashTableNames;
    int                 NbVar;
    int                 NbBindVar;

    WireLoadSelect = LibGetWireLoadSelectFromName (GnlLib, (char *)NULL);
    G_WireLoad = LibGetWireLoadFromAreaAndWireLS (WireLoadSelect,
                                                GnlNetworkNLDelayArea (Nw), GnlLib);
    G_GnlLibc = GnlLib;

    /* Now scanning all kept components to add their pin capacitance */
    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT)ComponentI;

        /* There is a Gnl associated so it is not linked to a cell */
        if (GnlUserComponentGnlDef (UserCompoI))
            continue;

        /* If no linked libc cell then is it a black box */
        if (!GnlUserComponentCellDef (UserCompoI))
            continue;

        Cell = (LIBC_CELL)GnlUserComponentCellDef (UserCompoI);
        if (LibCellDontUse (Cell))

```

```

{
    fprintf (stderr,
              " WARNING: Netlist using DONT USE cell [%s]\n",
              LibCellName(Cell));
}

/* Case where the component is linked to a sequential cell */
if (LibCellFFLatch (Cell))
{
{
    GnlComputeCapaFromUserCompo (Gnl, GnlLib, UserCompoI, Cell);
    continue;
}

/* Case where the component is linked to a 3 states */
if (LibCellWith3StatePin (Cell, &OutPin))
{
{
    GnlComputeCapaFromUserCompo (Gnl, GnlLib, UserCompoI, Cell);
    continue;
}

/* Case where the component is linked to a buffer. */
if (GnlCellIsBuffer (Cell))
{
{
    fprintf (stderr, "\n ERROR: all buffers not replaced.\n");
    exit (1);
}

fprintf (stderr,
          "\n ERROR: all combinational compo. nor replaced.\n");
exit (1);
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    if (!GnlVarFunction (VarI))
        continue;

    Function = GnlVarFunction (VarI);

    if (GnlComputeCapaFromKeptCompoOnNode (Gnl, GnlLib,
                                            GnlFunctionOnSet (Function)))
        return (GNL_MEMORY_FULL);
}

/* Then computing the wire capacitance for all the variables with */
/* a non null capacitance. */
NbVar = NbBindVar = 0;
HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        Fanout = (int)GnlVarHook (VarJ);
}

```

gnlverif.c

```
    if (Fanout)
        NbVar++;

    /* if 'VarJ' is not the input of a kept component then its */
    /* capa. is 0.0                                         */
    if (GnlVarOutCapa (VarJ) == 0.0)
        continue;

    NbBindVar++;

    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad,
                                                Length, GnlLib);

    SetGnlVarOutCapa (VarJ, GnlVarOutCapa (VarJ) + WireCapa);
}

}

return (GNL_OK);
}

/*-----
/* GnlComputeCapacitancesFromKeptCompoRec                         */
/*-----*/
GNL_STATUS GnlComputeCapacitancesFromKeptCompoRec (Nw, Gnl, GnlLib)
    GNL_NETWORK    Nw;
    GNL           Gnl;
    LIBC_LIB      GnlLib;
{
    int            i;
    BLIST          Components;
    GNL_COMPONENT ComponentI;
    GNL_USER_COMPONENT UserCompoI;
    GNL           GnlCompoI;

    /* Already processed.                                         */
    if (GnlTag (Gnl) == GnlNetworkTag (Nw))
        return (GNL_OK);

    SetGnlTag (Gnl, GnlNetworkTag (Nw));

    if (GnlComputeCapacitancesFromKeptCompo (Nw, Gnl, GnlLib))
        return (GNL_MEMORY_FULL);

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);

        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;

        UserCompoI = (GNL_USER_COMPONENT)ComponentI;
        if (!GnlUserComponentGnlDef (UserCompoI))
            continue;
```

```

GnlCompoI = GnlUserComponentGnlDef (UserCompoI);

    if (GnlComputeCapacitancesFromKeptCompoRec (Nw, GnlCompoI,
                                                GnlLib))
        return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*
 * GnlReplaceAllComponentsWithLinkLibCells
 */
/*
 * We modify the original GNL_USER_COMPONENT components according to
 * to their linking libc cell. For example a Component related to a
 * Flip-Flop will be replaced into GNL_SEQUENTIAL_COMPONENT and the same
 * for LATCHES, TRISTATES, ...
 * Other Components which are combinatorial are either linked to a
 * LIBC_CELL and then are replaced by the corresponding Boolean Function
 * or have no linked LIBC_CELL and then they are black box.
 */
GNL_STATUS GnlReplaceAllComponentsWithLinkLibCells (Nw, Gnl, GnlLib,
                                                    GetEquivDeriveCells)
{
    GNL_NETWORK Nw;
    GNL         Gnl;
    LIBC_LIB    GnlLib;
    int         GetEquivDeriveCells;

    /* Resetting the Tag for recursive traversal.
     * SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    if (GnlReplaceAllComponentsWithLinkLibCellsRec (Nw, Gnl, GnlLib, 1,
                                                    GetEquivDeriveCells))
        return (GNL_MEMORY_FULL);

    /* Resetting the Tag for recursive traversal.
     * SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
     if (GnlUpdateLocalVarDirRec (Nw, Gnl))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
 * GnlReplacePredefComponentsWithLinkLibCells
 */
/*
 * We modify the original GNL_USER_COMPONENT components according to
 * to their linking libc cell. For example a Component related to a
 * Flip-Flop will be replaced into GNL_SEQUENTIAL_COMPONENT and the same
 * for LATCHES, TRISTATES, ...
 * Other Components are not replaced.
 */
GNL_STATUS GnlReplacePredefComponentsWithLinkLibCells (Nw, Gnl, GnlLib)
{
    GNL_NETWORK Nw;

```

gnlverif.c

```
GNL          Gnl;
LIBC_LIB     GnlLib;
{
    int          GetEquivDeriveCells;

    /* Resetting the Tag for recursive traversal. */ 
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

    GetEquivDeriveCells = 1;
    return (GnlReplaceAllComponentsWithLinkLibCellsRec (Nw, Gnl, GnlLib,
                                                       0, GetEquivDeriveCells));
}

/*-----*/
/* GnlReplaceComponentsWithLinkLibCellsForNLDelayMapping */
/*-----*/
/* We modify the original GNL_USER_COMPONENT components according to */
/* to their linking libc cell. Only combinatorial Components will be */
/* replaced and others define as black box components. */
/*-----*/
GNL_STATUS GnlReplaceComponentsWithLinkLibCellsForNLDelayMapping (Nw,
                                                               Gnl, GnlLib)
{
    GNL_NETWORK  Nw;
    GNL          Gnl;
    LIBC_LIB     GnlLib;
}

/* Computing the netlist area for the NL delay model */ 
SetGnlNetworkNLDelayArea (Nw, 0.0);
if (GnlGetNLDelayAreaRec (Nw, Gnl, GnlLib))
    return (GNL_MEMORY_FULL);

/* Resetting the Tag for recursive traversal. */ 
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

/* we compute first the fanout of each var. */ 
GnlComputeMaxFanoutInNetworkInGnlRec (Nw, Gnl);

/* Resetting the Tag for recursive traversal. */ 
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

if (GnlReplaceComponentsWithLinkLibCellsForNLDelayMappingRec (Nw,
                                                               Gnl, GnlLib))
    return (GNL_MEMORY_FULL);

/* Resetting the Tag for recursive traversal. */ 
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
if (GnlUpdateLocalVarDirRec (Nw, Gnl))
    return (GNL_MEMORY_FULL);

/* Resetting the Tag for recursive traversal. */ 
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

/* We remove all components node reference if the component is not */
/* in a multi-fanout situation (e.g one of its sons is a multi-fanout*/ 
/* node). */ 
/* */
if (GnlRemoveCompoNotToKeepRec (Nw, Gnl, GnlLib))
```

gnlverif.c

```
    return (GNL_MEMORY_FULL);

/* Resetting the Tag for recursive traversal. */  
SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);

if (GnlComputeCapacitancesFromKeptCompoRec (Nw, Gnl, GnlLib))  
    return (GNL_MEMORY_FULL);

GnlPrintGnl (stderr, Gnl);

return (GNL_OK);
}

/*-----*/  
/* GnlReplaceOnlyCombinatorialComponentsWithLinkLibCells */  
/*-----*/  
/* We modify the original GNL_USER_COMPONENT components according to */  
/* to their linking libc cell. Only combinatorial Components will be */  
/* replaced and others define as black box components. */  
/*-----*/  
GNL_STATUS GnlReplaceOnlyCombinatorialComponentsWithLinkLibCells (Nw,  
                                                               Gnl, GnlLib)  
    GNL_NETWORK Nw;  
    GNL Gnl;  
    LIBC_LIB GnlLib;  
{  
  
    /* If we target for an optimization of arrival time based on the */  
    /* NL delay model then we do a special link. */  
    if (GnlEnvCriterion () == GNL_NL_DELAY)  

```

gnlverif.c

```
GNL          Gnl;
BLIST        *PIs;
BLIST        *POs;
BLIST        *PIOs;
BLIST        *DffsO;
BLIST        *LatchsO;
BLIST        *TristatesO;
{
    int                  i;
    int                  j;
    BLIST                HashTableNames;
    BLIST                BucketI;
    GNL_VAR              VarJ;
    GNL_VAR              OutPut;
    GNL_VAR              OutPutBar;
    GNL_COMPONENT        ComponentI;
    GNL_USER_COMPONENT   UserCompo;
    GNL_TRISTATE_COMPONENT TriCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_NODE              NewNode;
    GNL_NODE              NodeNot;
    GNL_VAR              Input;
    GNL_FUNCTION         NewFunction;
    GNL_VAR              NewVar;

    if (BListCreateWithSize (1, PIs))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, POs))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, PIOs))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, DffsO))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, LatchsO))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, TristatesO))
        return (GNL_MEMORY_FULL);

    HashTableNames = GnlHashNames (Gnl);
    for (i=0; i<BListSize (HashTableNames); i++)
    {
        BucketI = (BLIST)BListElt (HashTableNames, i);
        for (j=0; j<BListSize (BucketI); j++)
        {
            VarJ = (GNL_VAR)BListElt (BucketI, j);
            if (GnlVarRangeSize (VarJ) != 1)
                continue;
            switch (GnlVarDir (VarJ)) {
            case GNL_VAR_INPUT:
                if (BListAddElt (*PIs, (int)VarJ))
                    return (GNL_MEMORY_FULL);
                break;

            case GNL_VAR_OUTPUT:
                if (BListAddElt (*POs, (int)VarJ))
                    return (GNL_MEMORY_FULL);
                break;
            }
        }
    }
}
```

```

        break;

    case GNL_VAR_INOUT:
        if (BListAddElt (*PIOS, (int)VarJ))
            return (GNL_MEMORY_FULL);
        break;

    default:
        break;
    }
}

for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);
    switch (GnlComponentType (ComponentI)) {
        case GNL_USER_COMPO:
            UserCompo = (GNL_USER_COMPONENT)ComponentI;
            if (GnlUserComponentGnlDef (UserCompo))
                fprintf (stderr,
                    " ERROR: %s cannot verify hiearchical design\n",
                    GNL_TOOL_NAME);
            else
                fprintf (stderr,
                    " ERROR: %s cannot verify design with black box\n",
                    GNL_TOOL_NAME);
            exit (1);
            break;

        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
            Input = GnlSequentialCompoInput (SeqCompo);
            OutPut = GnlSequentialCompoOutput (SeqCompo);
            OutPutBar = GnlSequentialCompoOutputBar (SeqCompo);

            /* we create a function for the Q bar of the seq      */
            /* component.                                         */
            if (OutPut && OutPutBar)
            {
                if (GnlCreateNodeForVar (Gnl, OutPut, &NewNode))
                    return (GNL_MEMORY_FULL);
                if (GnlCreateNodeNot (Gnl, NewNode, &NodeNot))
                    return (GNL_MEMORY_FULL);

                if (GnlFunctionCreate (Gnl, OutPutBar, NULL,
                                      &NewFunction))
                    return (GNL_MEMORY_FULL);
                SetGnlVarFunction (OutPutBar, NewFunction);
                SetGnlFunctionOnSet (NewFunction, NodeNot);
            }
            else
                if (!OutPut)
                    { /* only output bar.                         */
                      OutPut = OutPutBar;
                    }
    }
}

```

gnlverif.c

```
if (GnlSeqComponentIsDFF (SeqCompo))
{
    if (BListAddElt (*DffsO, (int)OutPut))
        return (GNL_MEMORY_FULL);
}
else
{
    if (BListAddElt (*LatchsO, (int)OutPut))
        return (GNL_MEMORY_FULL);
}
break;

case GNL_TRISTATE_COMPO:
    TriCompo = (GNL_TRISTATE_COMPONENT)ComponentI;
    OutPut = GnlTriStateOutput (TriCompo);
    if (BListAddElt (*TristatesO, (int)OutPut))
        return (GNL_MEMORY_FULL);

break;

default:
    fprintf (stderr,
    " ERROR: %s aborted because of unknown component type\n",
    GNL_TOOL_NAME);
    exit (1);
}

return (GNL_OK);
}

/*-----
/* GnlCleanVarForVerification
/*-----*/
void GnlCleanVarForVerification (Gnl)
    GNL          Gnl;
{
    int          i;
    int          j;
    BLIST        BucketI;
    GNL_VAR      VarJ;
    BLIST        HashTableNames;

HashTableNames = GnlHashNames (Gnl);
for (i=0; i<BListSize (HashTableNames); i++)
{
    BucketI = (BLIST)BListElt (HashTableNames, i);
    for (j=0; j<BListSize (BucketI); j++)
    {
        VarJ = (GNL_VAR)BListElt (BucketI, j);
        SetGnlVarHook (VarJ, NULL);
        SetGnlVarDistance (VarJ, 0);
    }
}
}
```

gnlverif.c

```
/*-----*/
/* GnlGetVarCommonStringSize */
*/
/*-----*/
int GnlGetVarCommonStringSize (Var1, Var2)
    GNL_VAR Var1;
    GNL_VAR Var2;
{
    int          L1;
    int          L2;
    int          Common;
    int          i;
    int          Min;

    if (!strcmp (GnlVarName (Var1), GnlVarName (Var2)))
        return (1000000);

    L1 = strlen (GnlVarName (Var1));
    L2 = strlen (GnlVarName (Var2));
    Min = (L1 < L2 ? L1 : L2);

    Common = 0;
    for (i=0; i<Min; i++)
    {
        if (GnlVarName (Var1) [i] == GnlVarName (Var2) [i])
            Common++;
    }

    return (Common);
}

/*-----*/
/* GnlPrintListLinkedVar */
/*
void GnlPrintListLinkedVar (File, List1, List2)
    FILE          *File;
    BLIST         List1;
    BLIST         List2;
{
    int          i;
    GNL_VAR      Var1;
    GNL_VAR      Var2;

    for (i=0; i<BListSize (List1); i++)
    {
        Var1 = (GNL_VAR)BListElt (List1, i);
        Var2 = (GNL_VAR)BListElt (List2, i);
        fprintf (File, " ");
        GnlPrintFormatSignalName (File, GnlVarName (Var1), 20);
        fprintf (File, " <-> ");
        GnlPrintFormatSignalName (File, GnlVarName (Var2), 20);
        fprintf (File, "\n");
    }
    fprintf (File, "\n");
}
```

gnlverif.c

```
}

/*-----*/
/* GnlSortClosestNames */
/*-----*/
GNL_STATUS GnlSortClosestNames (List1, List2)
    BLIST    List1;
    BLIST    List2;
{
    int             i;
    int             j;
    GNL_VAR        Var1;
    GNL_VAR        Var2;
    int             Distance;
    BLIST          NewList;
    BLIST          Listvar;
    int             Found;
    GNL_VAR        VarAssoc;
    int             BestIndex;

    for (i=0; i<BListSize (List1); i++)
    {
        Var1 = (GNL_VAR)BListElt (List1, i);
        if (GnlVarHook (Var1))
        {
            VarAssoc = (GNL_VAR)GnlVarHook (Var1);
            /* we look for this var assoc in 'List2' and remove it */
            Found = 0;
            for (j=0; j<BListSize (List2); j++)
            {
                Var2 = (GNL_VAR)BListElt (List2, j);
                if (VarAssoc == Var2)
                {
                    Found = 1;
                    break;
                }
            }
            if (Found)
                BListDelInsert (List2, j+1);
            else
            {
                fprintf (stderr,
" ERROR: Var <%s> will be associated to several Variables: %s, ... \n",
                    GnlVarName (Var1), GnlVarName (VarAssoc));
                exit (1);
            }
        continue;
    }

    for (j=0; j<BListSize (List2); j++)
    {
        Var2 = (GNL_VAR)BListElt (List2, j);
        Distance = GnlGetVarCommonStringSize (Var1, Var2);
```

```

        if (Distance > GnlVarDistance (Var1))
        {
            SetGnlVarDistance (Var1, Distance);
            VarAssoc = Var2;
            BestIndex = j;
        }
    }

    SetGnlVarHook (Var1, VarAssoc);
    BListDelInsert (List2, BestIndex+1);
}

if (BListSize (List2) != 0)
{
    fprintf (stderr,
        " ERROR: Some variables not associated in the second netlist\n");
    exit (1);
}

for (i=0; i<BListSize (List1); i++)
{
    Var1 = (GNL_VAR)BListElt (List1, i);
    if (BListAddElt (List2, (int)GnlVarHook (Var1)))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*-----
/* GnlModifyGnlInterfaceForVerif
/*-----*/
GNL_STATUS GnlModifyGnlInterfaceForVerif (Gnl, PIs, POs, PIOs, DffsO,
                                         LatchsO, TristatesO)
{
    GNL          Gnl;
    BLIST        PIs;
    BLIST        POs;
    BLIST        PIOs;
    BLIST        DffsO;
    BLIST        LatchsO;
    BLIST        TristatesO;
{
    BLIST        Inputs;
    BLIST        Outputs;
    BLIST        NewList;
    int          i;
    GNL_VAR     VarI;
    GNL_VAR     Input;
    GNL_VAR     OutPut;
    GNL_VAR     OutPutBar;
    GNL_COMPONENT ComponentI;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_TRISTATE_COMPONENT TriCompo;

    Inputs = GnlInputs (Gnl);
    Outputs = GnlOutputs (Gnl);
}

```

gnlverif.c

```
BSize (Inputs) = 0;
BSize (Outputs) = 0;

for (i=0; i<BListSize (PIs); i++)
{
    VarI = (GNL_VAR)BListElt (PIs, i);
    if (BListAddElt (Inputs, (int)VarI))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (PIOs); i++)
{
    VarI = (GNL_VAR)BListElt (PIOs, i);
    if (BListAddElt (Inputs, (int)VarI))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (POs); i++)
{
    VarI = (GNL_VAR)BListElt (POs, i);
    if (GnlVarRangeSize (VarI) == 1)
    {
        if (BListAddElt (Outputs, (int)VarI))
            return (GNL_MEMORY_FULL);
    }
}
for (i=0; i<BListSize (PIOs); i++)
{
    VarI = (GNL_VAR)BListElt (PIOs, i);
    if (GnlVarRangeSize (VarI) == 1)
    {
        if (BListAddElt (Outputs, (int)VarI))
            return (GNL_MEMORY_FULL);
    }
}

GnlResetVarHook (Gnl);

for (i=0; i<BListSize (GnlComponents (Gnl)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);
    switch (GnlComponentType (ComponentI)) {
        case GNL_SEQUENTIAL_COMPO:
            SeqCompo = (GNL_SEQUENTIAL_COMPONENT)ComponentI;
            OutPut = GnlSequentialCompoOutput (SeqCompo);
            OutPutBar = GnlSequentialCompoOutputBar (SeqCompo);
            if (OutPut)
            {
                SetGnlVarHook (OutPut,
                                GnlSequentialCompoInput (SeqCompo));
            }
            else /* Only output bar */
            {
                SetGnlVarHook (OutPutBar,
                                GnlSequentialCompoInput (SeqCompo));
            }
            break;
    }
}
```

```

        case GNL_TRISTATE_COMPO:
            TriCompo = (GNL_TRISTATE_COMPONENT)ComponentI;
            OutPut = GnlTriStateOutput (TriCompo);
            SetGnlVarHook (OutPut, GnlTriStateInput (TriCompo));
            break;

        default:
            fprintf (stderr, " ERROR: unknown component\n");
            exit (1);
    }

for (i=0; i<BListSize (DffsO); i++)
{
    VarI = (GNL_VAR)BListElt (DffsO, i);
    if (BListAddElt (Inputs, (int)VarI))
        return (GNL_MEMORY_FULL);
    Input = (GNL_VAR)GnlVarHook (VarI);
    if (BListAddElt (Outputs, (int)Input))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (LatchsO); i++)
{
    VarI = (GNL_VAR)BListElt (LatchsO, i);
    if (BListAddElt (Inputs, (int)VarI))
        return (GNL_MEMORY_FULL);
    Input = (GNL_VAR)GnlVarHook (VarI);
    if (BListAddElt (Outputs, (int)Input))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (TristatesO); i++)
{
    VarI = (GNL_VAR)BListElt (TristatesO, i);
    if (BListAddElt (Inputs, (int)VarI))
        return (GNL_MEMORY_FULL);
    Input = (GNL_VAR)GnlVarHook (VarI);
    if (BListAddElt (Outputs, (int)Input))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}

/*
 * GnlReplaceBuffer
 */
/*
 * Replaces buffers components by a new function: outbuffer = inbuffer
 */
GNL_STATUS GnlReplaceBuffer (Gnl)
    GNL          Gnl;
{
    BLIST          Components;

```

gnlverif.c

```
int i;
GNL_COMPONENT ComponentI;
GNL_BUF_COMPONENT BufCompoI;
GNL_VAR Output;
GNL_VAR Input;
GNL_NODE NewNode;
GNL_FUNCTION NewFunction;

Components = GnlComponents (Gnl);
if (!Components)
    return (GNL_OK);

for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_BUF_COMPO)
        continue;

    BufCompoI = (GNL_BUF_COMPONENT)ComponentI;

    Output = GnlBufOutput (BufCompoI);
    Input = GnlBufInput (BufCompoI);

    /* if there is already a function associated then it is the case*/
    /* of a multi-source signal. We abort for now. */
    if (GnlVarFunction (Output))
    {
        fprintf (stderr, " ERROR: <%s> is a multi-source signal\n",
                 GnlVarName (Output));
        exit (1);
    }

    if (GnlFunctionCreate (Gnl, Output, NULL, &NewFunction))
        return (GNL_MEMORY_FULL);
    SetGnlVarFunction (Output, NewFunction);

    if (GnlCreateNodeForVar (Gnl, Input, &NewNode))
        return (GNL_MEMORY_FULL);

    SetGnlFunctionOnSet (NewFunction, NewNode);
    if (BListAddElt (GnlFunctions(Gnl), (int)Output))
        return (GNL_MEMORY_FULL);

    BListDelInsert (Components, i+1);
    i--;

    GnlFreeBufComponent (BufCompoI);
}

return (GNL_OK);
}

/*-----
/* GnlPrintPortAssociationsFile
/*-----*/
void GnlPrintPortAssociationsFile (OutAssocFile, TopGnl1, TopGnl2)
```

gnlverif.c

```
FILE *OutAssocFile;
GNL TopGnl1;
GNL TopGnl2;
{

    fprintf (OutAssocFile, " File automatically generated by %s %s\n",
             GNL_TOOL_NAME, GNL_MAPIT_VERSION);
    fprintf (OutAssocFile, " Date:");
    GnlPrintDate (OutAssocFile);
    fprintf (OutAssocFile, "\n");
    fprintf (OutAssocFile, "\n");
    fprintf (OutAssocFile, " Ports Associations between '%s' and '%s'.\n",
             GnlEnvInput (), GnlEnvReference ());
    fprintf (OutAssocFile, "\n");
    fprintf (OutAssocFile, " Inputs Associations\n");
    fprintf (OutAssocFile, " ----- \n");
    GnlPrintListLinkedVar (OutAssocFile, GnlInputs (TopGnl1),
                           GnlInputs (TopGnl2));
    fprintf (OutAssocFile, " Outputs Associations\n");
    fprintf (OutAssocFile, " ----- \n");
    GnlPrintListLinkedVar (OutAssocFile, GnlOutputs (TopGnl1),
                           GnlOutputs (TopGnl2));
}

/*-----
/* GnlAnalyzeAssociationLine
*/
/*-----*/
GNL_STATUS GnlAnalyzeAssociationLine (SLine, Name1, Name2)
{
    char *SLine;
    char **Name1;
    char **Name2;
{
    char Temp1[2000];
    char Temp2[2000];
    int i;
    int j;

    *Name1 = *Name2 = NULL;
    if (strlen (SLine) < 2)
        return (GNL_OK);

    i = 1;
    while (SLine[i] != '\0')
    {
        Temp1[i-1] = SLine[i];
        if (Temp1[i-1] == ' ')
        {
            Temp1[i-1] = '\0';
            break;
        }
        i++;
    }

    if (Temp1[i-1] != '\0')
        return (GNL_OK);
}
```

gnlverif.c

```
while ((SLine[i] == ' ') && (SLine[i] != '\0')) i++;

if (SLine[i] == '\0')
    return (GNL_OK);

if ((SLine[i] != '<') && (SLine[i+1] != '-') && (SLine[i+2] != '-') &&
    (SLine[i+3] != '>'))
    return (GNL_OK);

i = i+4;

if (SLine[i] == '\0')
    return (GNL_OK);

while ((SLine[i] == ' ') && (SLine[i] != '\0')) i++;

if (SLine[i] == '\0')
    return (GNL_OK);

j=0;
while ((SLine[i] != ' ') && (SLine[i] != '\0'))
{
    Temp2[j] = SLine[i];
    j++;
    i++;
}
Temp2[j] = '\0';

if ((*Name1 = (char*)calloc (strlen (Temp1)+1, sizeof (char))) == NULL)
    return (GNL_MEMORY_FULL);
strcpy (*Name1, Temp1);

if ((*Name2 = (char*)calloc (strlen (Temp2)+1, sizeof (char))) == NULL)
    return (GNL_MEMORY_FULL);
strcpy (*Name2, Temp2);

return (GNL_OK);
}

/*-----*/
/* GnlReadFileAssoc */
/*-----*/
GNL_STATUS GnlReadFileAssoc (InputAssocFileName, ListInputs1,
                             ListOutputs1, ListInputs2, ListOutputs2)
char           *InputAssocFileName;
BLIST          *ListInputs1;
BLIST          *ListOutputs1;
BLIST          *ListInputs2;
BLIST          *ListOutputs2;
{
FILE           *OutFile;
char           SLine[2000];
int            Index;
int            Mode;
char           *Name1;
```

```

char *Name2;

if ((OutFile = fopen (InputAssocFileName, "r")) == NULL)
{
    fprintf (stderr, " ERROR: cannot open file '%s'\n",
             InputAssocFileName);
    exit (1);
}

if (BListCreateWithSize (1, ListInputs1))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (1, ListOutputs1))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (1, ListInputs2))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (1, ListOutputs2))
    return (GNL_MEMORY_FULL);

Mode = 0;
Index = 0;
while ((SLine[Index] = getc (OutFile)) != EOF)
{
    if (SLine[Index] == '\n')
    {
        SLine[Index] = '\0';
        Index = 0;

        if (!strcmp (SLine, " Inputs Associations"))
            Mode = 1;
        if (!strcmp (SLine, " Outputs Associations"))
            Mode = 2;

        if (GnlAnalyzeAssociationLine (SLine, &Name1, &Name2))
            return (GNL_MEMORY_FULL);

        if (Name1)
        {
            if (Mode == 1)
            {
                if (BListAddElt (*ListInputs1, (int)Name1))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (*ListInputs2, (int)Name2))
                    return (GNL_MEMORY_FULL);
            }
            else if (Mode == 2)
            {
                if (BListAddElt (*ListOutputs1, (int)Name1))
                    return (GNL_MEMORY_FULL);
                if (BListAddElt (*ListOutputs2, (int)Name2))
                    return (GNL_MEMORY_FULL);
            }
        }
    }
    Index++;
}

```

```

        }

    fclose (OutFile);

    return (GNL_OK);
}

/*-----
/* GnlModifyGnlInterfaceWithUserAssoc
/*-----*/
void GnlModifyGnlInterfaceWithUserAssoc (Gnl1, Gnl2, ListInputs1,
                                         ListOutputs1, ListInputs2,
                                         ListOutputs2)
{
    GNL          Gnl1;
    GNL          Gnl2;
    BLIST        ListInputs1;
    BLIST        ListOutputs1;
    BLIST        ListInputs2;
    BLIST        ListOutputs2;
{
    int          i;
    int          j;
    char         *InputNameI;
    int          Found;
    GNL_VAR     InputJ;

    for (i=0; i<BListSize (ListInputs1); i++)
    {
        InputNameI = (char*)BListElt (ListInputs1, i);
        Found = 0;
        for (j=0; j<BListSize (GnlInputs (Gnl1)); j++)
        {
            InputJ = (GNL_VAR)BListElt (GnlInputs (Gnl1), j);
            if (!strcmp (InputNameI, GnlVarName (InputJ)))
            {
                Found = 1;
                free (InputNameI);
                BListElt (ListInputs1, i) = (int)InputJ;
                BListDelInsert (GnlInputs (Gnl1), j+1);
                break;
            }
        }
        if (!Found)
        {
            fprintf (stderr,
                     " ERROR: Input signal '%s' not found in '%s'\n",
                     InputNameI, GnlName (Gnl1));
            exit (1);
        }
    }

    if (BListSize (GnlInputs (Gnl1)) != 0)
    {
        fprintf (stderr, " ERROR: unassociated input signals: ");
        for (j=0; j<BListSize (GnlInputs (Gnl1)); j++)

```

```

    {
        InputJ = (GNL_VAR)BListElt (GnlInputs (Gnl1), j);
        fprintf (stderr, "%s ", GnlVarName (InputJ));
    }
    fprintf (stderr, "\n");
    exit (1);
}

BListQuickDelete (&GnlInputs (Gnl1));
SetGnlInputs (Gnl1, ListInputs1);

for (i=0; i<BListSize (ListInputs2); i++)
{
    InputNameI = (char*)BListElt (ListInputs2, i);
    Found = 0;
    for (j=0; j<BListSize (GnlInputs (Gnl2)); j++)
    {
        InputJ = (GNL_VAR)BListElt (GnlInputs (Gnl2), j);
        if (!strcmp (InputNameI, GnlVarName (InputJ)))
        {
            Found = 1;
            free (InputNameI);
            BListElt (ListInputs2, i) = (int)InputJ;
            BListDelInsert (GnlInputs (Gnl2), j+1);
            break;
        }
    }
    if (!Found)
    {
        fprintf (stderr,
                  " ERROR: Input signal '%s' not found in '%s'\n",
                  InputNameI, GnlName (Gnl2));
        exit (1);
    }
}

if (BListSize (GnlInputs (Gnl2)) != 0)
{
    fprintf (stderr, " ERROR: unassociated input signals: ");
    for (j=0; j<BListSize (GnlInputs (Gnl2)); j++)
    {
        InputJ = (GNL_VAR)BListElt (GnlInputs (Gnl2), j);
        fprintf (stderr, "%s ", GnlVarName (InputJ));
    }
    fprintf (stderr, "\n");
    exit (1);
}

BListQuickDelete (&GnlInputs (Gnl2));
SetGnlInputs (Gnl2, ListInputs2);

for (i=0; i<BListSize (ListOutputs1); i++)
{
    InputNameI = (char*)BListElt (ListOutputs1, i);
    Found = 0;
    for (j=0; j<BListSize (GnlOutputs (Gnl1)); j++)

```

gnlverif.c

```
{  
    InputJ = (GNL_VAR)BListElt (GnlOutputs (Gnl1), j);  
    if (!strcmp (InputNameI, GnlVarName (InputJ)))  
    {  
        Found = 1;  
        free (InputNameI);  
        BListElt (ListOutputs1, i) = (int)InputJ;  
        BListDelInsert (GnlOutputs (Gnl1), j+1);  
        break;  
    }  
}  
if (!Found)  
{  
    fprintf (stderr,  
            " ERROR: Output signal '%s' not found in '%s'\n",  
            InputNameI, GnlName (Gnl1));  
    exit (1);  
}  
}  
  
BListQuickDelete (&GnlOutputs (Gnl1));  
SetGnlOutputs (Gnl1, ListOutputs1);  
  
for (i=0; i<BListSize (ListOutputs2); i++)  
{  
    InputNameI = (char*)BListElt (ListOutputs2, i);  
    Found = 0;  
    for (j=0; j<BListSize (GnlOutputs (Gnl2)); j++)  
    {  
        InputJ = (GNL_VAR)BListElt (GnlOutputs (Gnl2), j);  
        if (!strcmp (InputNameI, GnlVarName (InputJ)))  
        {  
            Found = 1;  
            free (InputNameI);  
            BListElt (ListOutputs2, i) = (int)InputJ;  
            BListDelInsert (GnlOutputs (Gnl2), j+1);  
            break;  
        }  
    }  
    if (!Found)  
    {  
        fprintf (stderr,  
                " ERROR: Output signal '%s' not found in '%s'\n",  
                InputNameI, GnlName (Gnl2));  
        exit (1);  
    }  
}  
  
BListQuickDelete (&GnlOutputs (Gnl2));  
SetGnlOutputs (Gnl2, ListOutputs2);  
}  
  
/*-----*/  
/* CmpGnlVarId */  
/*-----*/
```

gnlverif.c

```
int CmpGnlVarId (Var1, Var2)
    GNL_VAR      *Var1;
    GNL_VAR      *Var2;
{
    if (GnlVarId (*Var1) < GnlVarId (*Var2))
        return (-1);

    if (GnlVarId (*Var1) == GnlVarId (*Var2))
        return (0);

    return (1);
}

/*-----
/* GnlEcVerification
/*-----*/
GNL_STATUS GnlEcVerification ()
{
    int                      i;
    GNL                      Gnl;
    GNL                      GnlI;
    BLIST                     ListGnls1;
    BLIST                     ListGnls2;
    LIBC_LIB                  GnlLib;
    GNL_LIB                  HGnlLib;
    int                      Done;
    FILE                      *OutFile;
    GNL_STATUS                GnlStatus;
    GNL                      TopGnl1;
    GNL                      TopGnl2;
    GNL_NETWORK               GlobalNetwork1;
    GNL_NETWORK               GlobalNetwork2;
    BLIST                     PIs1;
    BLIST                     POS1;
    BLIST                     PIOs1;
    BLIST                     DffsO1;
    BLIST                     LatchsO1;
    BLIST                     TristatesO1;
    BLIST                     PIs2;
    BLIST                     POS2;
    BLIST                     PIOs2;
    BLIST                     DffsO2;
    BLIST                     LatchsO2;
    BLIST                     TristatesO2;
    int                      MaxNbNodes;
    int                      NbMayErrors;
    BLIST                     Gnl1MayErrors;
    BLIST                     Gnl2MayErrors;
    int                      NbErrors;
    BLIST                     Gnl1Errors;
    BLIST                     Gnl2Errors;
    BLIST                     GnlBadPatterns;
    GNL_VAR                  Var1;
    GNL_VAR                  Var2;
    BLIST                     ListAssocIn;
    BLIST                     ListAssocOut;
```

gnlverif.c

```
FILE *OutAssocFile;
FILE *InputAssocFile;
BLIST ListInputs1;
BLIST ListInputs2;
BLIST ListOutputs1;
BLIST ListOutputs2;

if (GnlEnvInput())
{
    if ((OutFile = fopen (GnlEnvInput(), "r")) == NULL)
    {
        fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
                 GnlEnvInput());
        return (GNL_CANNOT_OPEN_OUTFILE);
    }
    fclose (OutFile);
}

if (GnlEnvReference())
{
    if ((OutFile = fopen (GnlEnvReference(), "r")) == NULL)
    {
        fprintf (stderr, " ERROR: Cannot Open Reference File '%s'\n",
                 GnlEnvReference());
        return (GNL_CANNOT_OPEN_OUTFILE);
    }
    fclose (OutFile);
}

if ((OutFile = fopen (GnlEnvLib(), "r")) == NULL)
{
    fprintf (stderr, " ERROR: Cannot Open Library File '%s'\n",
             GnlEnvLib());
    return (GNL_CANNOT_OPEN_LIBRARY);
}
fclose (OutFile);

if (GnlEnvInputFormat() != GNL_INPUT_VLG)
{
    fprintf (stderr,
             " ERROR: Verification must be done on Verilog netlist input\n");
    return (GNL_CANNOT_OPEN_INPUTFILE);
}

fprintf (stderr, "\n Reading Verilog file [%s] ... \n",
         GnlEnvInput());
if (GnlRead (GnlEnvInput(), &ListGnls1))
{
    fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
             GnlEnvInput());
    return (GNL_CANNOT_OPEN_INPUTFILE);
}
fprintf (stderr, " Verilog Netlist Analyzed\n");

fprintf (stderr, "\n Reading Verilog file [%s] ... \n",
         GnlEnvReference());
```

gnlverif.c

```
if (GnlRead (GnlEnvReference (), &ListGnls2))
{
    fprintf (stderr, " ERROR: Cannot Open Input File '%s'\n",
             GnlEnvReference ());
    return (GNL_CANNOT_OPEN_INPUTFILE);
}
fprintf (stderr, " Verilog Netlist Analyzed\n");
fprintf (stderr, "\n");

/* Sort and prints the list 'ListGnls' by putting first the top          */
/* level modules
SortTopLevelModules (ListGnls1);
SortTopLevelModules (ListGnls2);

if (!GnlEnvTop ())
{
    GnI = (GNL)BListElt (ListGnls1, 0);
    TopGnI = GnI;
    fprintf (stderr,
             " WARNING: Top Level Module is [%s] by default\n",
             GnlName (TopGnI));
}
else
{
    /* Taking the top level module from the user                      */
    TopGnI = NULL;
    for (i=0; i<BListSize (ListGnls1); i++)
    {
        GnI = (GNL)BListElt (ListGnls1, i);
        if (!strcmp (GnlName (GnI), GnlEnvTop ()))
        {
            TopGnI = GnI;
            break;
        }
    }
}
if (TopGnI == NULL)
{
    fprintf (stderr, " ERROR: cannot find top-level module [%s]\n",
             GnlEnvTop ());
    return (GNL_CANNOT_FIND_TOP_LEVEL);
}

if (!GnlEnvTop ())
{
    GnI = (GNL)BListElt (ListGnls2, 0);
    TopGnI = GnI;
    fprintf (stderr,
             " WARNING: Top Level Module is [%s] by default\n",
             GnlName (TopGnI));
}
else
{
    /* Taking the top level module from the user                      */
    TopGnI = NULL;
    for (i=0; i<BListSize (ListGnls2); i++)
    {
```

```

GnlI = (GNL)BListElt (ListGnls2, i);
if (!strcmp (GnlName (GnlI), GnlEnvTop()))
{
    TopGnl2 = GnlI;
    break;
}
}
if (TopGnl2 == NULL)
{
    fprintf (stderr, " ERROR: cannot find top-level module [%s]\n",
             GnlEnvTop());
    return (GNL_CANNOT_FIND_TOP_LEVEL);
}

/* we create the Global network. */
if (GnlCreateNetwork (TopGnl1, &GlobalNetwork1))
    return (GNL_MEMORY_FULL);
if (GnlCreateNetwork (TopGnl2, &GlobalNetwork2))
    return (GNL_MEMORY_FULL);

/* Reading and building the technology library */
fprintf (stderr, "\n Reading Library [%s] ... \n",
         GnlEnvLib());
if (LibRead (GnlEnvLib(), &GnlLib))
{
    fprintf (stderr, " ERROR: Library analysis aborted\n");
    return (GNL_ERROR_LIBRARY_READING);
}
fprintf (stderr, " Library analyzed\n");

/* Checking correctness of the network 'GlobalNetwork'. */
if ((GnlStatus = GnlCheckNetwork (GlobalNetwork1)))
    return (GnlStatus);
if ((GnlStatus = GnlCheckNetwork (GlobalNetwork2)))
    return (GnlStatus);

fprintf (stderr,
        "\n Checking Hierachical interfaces from top module [%s]\n",
        GnlName (TopGnl1));

/* We verify the correct connection between user components and
/* their definitions and update some fields (RefCount). */
if (GnlUpdateNCheckHierarchyInterface (GlobalNetwork1, TopGnl1,
                                         ListGnls1))
    return (GNL_MEMORY_FULL);
if (GnlUpdateNCheckHierarchyInterface (GlobalNetwork2, TopGnl2,
                                         ListGnls2))
    return (GNL_MEMORY_FULL);

/* Performing full flattening for both netlists */
fprintf (stderr,
        "\n Performing Flattening of modules in [%s] (file=' %s')\n",
        GnlName (TopGnl1), GnlEnvInput());
if (GnlFlattenFullGnlComponents (GlobalNetwork1, TopGnl1, ListGnls1))
    return (GNL_MEMORY_FULL);
fprintf (stderr, " Flattening done\n");

```

gnlverif.c

```
fprintf (stderr,
        "\n Performing Flattening of modules in [%s] (file='%s')\n",
        GnlName (TopGnl2), GnlEnvReference ());
if (GnlFlattenFullGnlComponents (GlobalNetwork2, TopGnl2, ListGnls2))
    return (GNL_MEMORY_FULL);
fprintf (stderr, " Flattening done\n");

/* We remove the buffers from the two netlists */
if (GnlReplaceBuffer (TopGnl1))
    return (GNL_MEMORY_FULL);
if (GnlReplaceBuffer (TopGnl2))
    return (GNL_MEMORY_FULL);

/* We link each component of all the Gnls in the network with the      */
/* libc cells of same name.                                              */
fprintf (stderr, "\n");
fprintf (stderr, " Linking Components With Libc Cells\n");
if (GnlLinkLib (GlobalNetwork1, TopGnl1, GnlLib))
    return (GNL_MEMORY_FULL);
if (GnlLinkLib (GlobalNetwork2, TopGnl2, GnlLib))
    return (GNL_MEMORY_FULL);

/* We modify the original GNL_USER_COMPONENT components according to */
/* to their linking libc cell.                                         */
fprintf (stderr, "\n");
fprintf (stderr, " Building Cell Functionalities..\n");
if (GnlReplaceAllComponentsWithLinkLibCells (GlobalNetwork1, TopGnl1,
                                             GnlLib, 0))
    return (GNL_MEMORY_FULL);
if (GnlReplaceAllComponentsWithLinkLibCells (GlobalNetwork2, TopGnl2,
                                             GnlLib, 0))
    return (GNL_MEMORY_FULL);

/* We extract the Primarys IOs of both netlists and outputs of Dffs, */
/* latches, Tri-states                                                 */
if (GnlExtractTypeOfSignalsInGnl (TopGnl1, &PIs1, &POs1, &PIOs1,
                                  &DffsO1, &LatchsO1, &TristatesO1))
    return (GNL_MEMORY_FULL);
if (GnlExtractTypeOfSignalsInGnl (TopGnl2, &PIs2, &POs2, &PIOs2,
                                  &DffsO2, &LatchsO2, &TristatesO2))
    return (GNL_MEMORY_FULL);

fprintf (stderr, " DESIGN: [%s] (file='%s')\n",
        GnlName (TopGnl1), GnlEnvInput ());
fprintf (stderr, "          PIs      = %6d\n", BListSize (PIs1));
fprintf (stderr, "          POs      = %6d\n", BListSize (POs1));
fprintf (stderr, "          PIOs     = %6d\n", BListSize (PIOs1));
fprintf (stderr, "          Dffs     = %6d\n", BListSize (DffsO1));
fprintf (stderr, "          Latches  = %6d\n",
        BListSize (LatchsO1));
fprintf (stderr, "          Tristates = %6d\n",
        BListSize (TristatesO1));
fprintf (stderr, "\n");

fprintf (stderr, " DESIGN: [%s] (file='%s')\n",
        GnlName (TopGnl2), GnlEnvReference ());
```

gnlverif.c

```
fprintf (stderr, "      PIs      = %6d\n", BListSize (PIs2));
fprintf (stderr, "      POs      = %6d\n", BListSize (POs2));
fprintf (stderr, "      PIOs     = %6d\n", BListSize (PIOs2));
fprintf (stderr, "      Dffs     = %6d\n", BListSize (DffsO2));
fprintf (stderr, "      Latches   = %6d\n",
         BListSize (LatchsO2));
fprintf (stderr, "      Tristates = %6d\n",
         BListSize (TristatesO2));
fprintf (stderr, "\n");

if (BListSize (PIs1) != BListSize (PIs2))
{
    fprintf (stderr, " ERROR: Number of PIs are different.\n");
    exit (1);
}
if (BListSize (POs1) != BListSize (POs2))
{
    fprintf (stderr, " ERROR: Number of POs are different.\n");
    exit (1);
}
if (BListSize (PIOs1) != BListSize (PIOs2))
{
    fprintf (stderr, " ERROR: Number of PIOs are different.\n");
    exit (1);
}
if (BListSize (DffsO1) != BListSize (DffsO2))
{
    fprintf (stderr, " ERROR: Number of Dffs are different.\n");
    exit (1);
}
if (BListSize (LatchsO1) != BListSize (LatchsO1))
{
    fprintf (stderr, " ERROR: Number of Latches are different.\n");
    exit (1);
}
if (BListSize (TristatesO1) != BListSize (TristatesO2))
{
    fprintf (stderr, " ERROR: Number of TriStates are different.\n");
    exit (1);
}

GnlCleanVarForVerification (TopGnl1);
GnlCleanVarForVerification (TopGnl2);

qsort (BListAddress (PIs1), BListSize (PIs1), sizeof(GNL_VAR),
       CmpGnlVarId);
qsort (BListAddress (PIOs1), BListSize (PIOs1), sizeof(GNL_VAR),
       CmpGnlVarId);

/* We sort the names in the two lists according to their distance */
GnlResetVarHook (TopGnl1);
GnlResetVarHook (TopGnl2);
if (GnlSortClosestNames (PIs1, PIs2))
    return (GNL_MEMORY_FULL);
if (GnlSortClosestNames (POs1, POs2))
    return (GNL_MEMORY_FULL);
if (GnlSortClosestNames (PIOs1, PIOs2))
```

gnlverif.c

```
    return (GNL_MEMORY_FULL);
if (GnlSortClosestNames (Dffs01, Dffs02))
    return (GNL_MEMORY_FULL);
if (GnlSortClosestNames (Latchs01, Latchs02))
    return (GNL_MEMORY_FULL);
if (GnlSortClosestNames (Tristates01, Tristates02))
    return (GNL_MEMORY_FULL);

if (GnlModifyGnlInterfaceForVerif (TopGnl1, PIs1, POs1, PIos1, Dffs01,
                                  Latchs01, Tristates01))
    return (GNL_MEMORY_FULL);

if (GnlModifyGnlInterfaceForVerif (TopGnl2, PIs2, POs2, PIos2, Dffs02,
                                  Latchs02, Tristates02))
    return (GNL_MEMORY_FULL);

MaxNbNodes = GnlEnvMaxBddNode();
if (MaxNbNodes == 0)
    MaxNbNodes = 500;

/* If the user asked for reading the ports associations in the      */
/* file then we take them into account.                                */
if (GnlEnvInputAssoc ())
{
    if (GnlReadFileAssoc (GnlEnvInputAssoc (), &ListInputs1,
                          &ListOutputs1, &ListInputs2, &ListOutputs2))
        return (GNL_MEMORY_FULL);

    GnlModifyGnlInterfaceWithUserAssoc (TopGnl1, TopGnl2,
                                       ListInputs1, ListOutputs1,
                                       ListInputs2, ListOutputs2);
}

fprintf (stderr, " Inputs Associations\n");
fprintf (stderr, " -----\n");
GnlPrintListLinkedVar (stderr, GnlInputs (TopGnl1),
                      GnlInputs (TopGnl2));

fprintf (stderr, " Outputs Associations\n");
fprintf (stderr, " -----\n");
GnlPrintListLinkedVar (stderr, GnlOutputs (TopGnl1),
                      GnlOutputs (TopGnl2));

/* If the user asked for printing the ports associations then we   */
/* do it.                                                       */
if (GnlEnvOutputAssoc ())
{
    if ((OutAssocFile = fopen (GnlEnvOutputAssoc (), "w")) == NULL)
    {
        fprintf (stderr,
                 " WARNING: Cannot Open Ports Associations File '%s'...\n",
                 GnlEnvOutputAssoc ());
    }
    else
    {
        fprintf (stderr,
```

```

        " Printing Ports Associations in file '%s'...\n",
        GnlEnvOutputAssoc()));

    GnlPrintPortAssociationsFile (OutAssocFile, TopGnl1, TopGnl2);
    fclose (OutAssocFile);
}

/* Creating list of inputs assoc */ 
if (BListCreate (&ListAssocIn))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (GnlInputs (TopGnl1)); i++)
{
    Var1 = (GNL_VAR)BListElt (GnlInputs (TopGnl1), i);
    Var2 = (GNL_VAR)BListElt (GnlInputs (TopGnl2), i);
    if (BListAddElt (ListAssocIn, (int)Var1))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListAssocIn, (int)Var2))
        return (GNL_MEMORY_FULL);
}

/* Creating list of outputs assoc */ 
if (BListCreate (&ListAssocOut))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (GnlOutputs (TopGnl1)); i++)
{
    Var1 = (GNL_VAR)BListElt (GnlOutputs (TopGnl1), i);
    Var2 = (GNL_VAR)BListElt (GnlOutputs (TopGnl2), i);
    if (BListAddElt (ListAssocOut, (int)Var1))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListAssocOut, (int)Var2))
        return (GNL_MEMORY_FULL);
}

fprintf (stderr, "\n");
fprintf (stderr, " Performing Equivalence Checking...\n");
if (GnlequivCheckGnl (TopGnl1, TopGnl2, ListAssocIn, ListAssocOut,
                      MaxNbNodes,
                      &NbMayErrors, &Gnl1MayErrors, &Gnl2MayErrors,
                      &NbErrors, &Gnl1Errors, &Gnl2Errors,
                      &GnlBadPatterns))
{
    fprintf (stderr, " ERROR: Critical Problem During
Verification\n");
    exit (1);
}
fprintf (stderr, "\n");
fprintf (stderr, " Verification done !\n");
fprintf (stderr, "\n");

return (GNL_OK);
}

/*----- EOF -----*/

```



APPENDIX E

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

```
        }
.gate      {
    return (C_GATE);
}
(nor1|not) {
    yylval.ival = C_NOT;
```

JCE92 U.S. PTO
09/152304
12/28/00

APPENDIX E

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

cparse.l

```
%{
/*-----*/
/*
/*      File:          cparsel.l
/*      Version:        1.1
/*      Modifications: -
/*      Documentation: -
/*
/*      Class: non
/*      Inheritance:
/*
/*-----*/
/*-----*/
#include <stdio.h>
#include <strings.h>
#include "c-tokens.h"

char      namebuf[256];      /* to store temporaly the string      */
extern int yylineno;

%}

NAME      [a-zA-Z_0-9][\$/\<\>.\0-9a-zA-Z_\[\]\(\)]*
CR        [\n]
BLANK    [\t]

%%
=         {
    return ((int) '=');
}

.model   {
    return (C_MODEL);
}

.end     {
    return (C_END);
}

.inputs  {
    return (C_INPUTS);
}

.outputs {
    return (C_OUTPUTS);
}

.gate    {
    return (C_GATE);
}

(nor1|not) {
    yyval.ival = C_NOT;
```

cparse.1

```
        return (C_NOT) ;
    }

(and2|and3|and4|and)      {
    yylval.ival = C_AND;
    return (C_AND);
}

(nand2|nand3|nand4|nand)  {
    yylval.ival = C_NAND;
    return (C_NAND);
}

(or2|or3|or4|or)         {
    yylval.ival = C_OR;
    return (C_OR);
}

(nor2|nor3|nor4|nor)     {
    yylval.ival = C_NOR;
    return (C_NOR);
}

(xor2|xor3|xor4|xor)    {
    yylval.ival = C_XOR;
    return (C_XOR);
}

(xnor2|xnor3|xnor4|xnor) {
    yylval.ival = C_XNOR;
    return (C_XNOR);
}

zero                      {
    yylval.ival = C_ZERO;
    return (C_ZERO);
}

{CR}                      {
    yylineno++;
}

{NAME}                    {
    strcpy (namebuf,yytext);
    yylval.sval = namebuf;
    return (C_NAME);
}

;

%%

/*-----*/
/* yywrap                                */
/*-----*/
yywrap ()
{

```

cparse.l

```
    return (1);
}

/*-----*/
```

File generated by E-Hubble 3.0

cparse.y

```
%{  
/*-----*/  
/*-----*/  
/* File: cparse.y */  
/* Version: 1.1 */  
/* Modifications: - */  
/* Documentation: - */  
/*-----*/  
/* Class: none */  
/* Inheritance: */  
/*-----*/  
  
#include <stdio.h>  
#include <strings.h>  
  
#include "blist.h"  
#include "gnl.h"  
  
/*-----*/  
/* ERROR MESSAGES */  
/*-----*/  
#define ERR_NEVER_DECLARED \  
    "# Error (%d): Signal <%s> is not declared"  
  
/*-----*/  
/* EXTERN */  
/*-----*/  
int yylineno;  
extern char* yytext;  
GNL_STATUS GnlStatus;  
  
/*-----*/  
/* GLOBAL VARIABLES */  
/*-----*/  
BLIST ListOfGnls; /* The main list storing all the GNLS */  
extern GNL CurrentGnl; /* The current Gnl/Module we process */  
GNL_VAR Var;  
GNL_VAR VarRight;  
int VarMsb; /* Msb of current Var. */  
int VarLsb; /* Lsb of current Var. */  
char *NewName;  
GNL_FUNCTION NewFunction;  
BLIST VarListSignal;  
BLIST ListDefParam;  
char *InstanceName;  
GNL_NODE CstNode;  
int TheGate;  
  
/*-----*/  
%}  
  
%union {  
    int ival;  
    char *sval;
```

cparse.y

```
}

%start c_nl

/* Types for non-terminals. */  
%type <ival>      gate

/* Types for terminals */  
%token   <ival>      C_OR  
%token   <ival>      C_NOR  
%token   <ival>      C_AND  
%token   <ival>      C_NAND  
%token   <ival>      C_XOR  
%token   <ival>      C_XNOR  
%token   <ival>      C_NOT  
%token   <ival>      C_MODEL  
%token   <ival>      C_INPUTS  
%token   <ival>      C_OUTPUTS  
%token   <ival>      C_END  
%token   <ival>      C_GATE  
%token   <ival>      C_ZERO

%token  <sval>  C_NAME

%%

/*-----*/  
c_nl : {  
    yylineno = 1; /* Initialize at 1rst line */  
    if (BListCreate (&ListOfGnls))  
        return (GNL_MEMORY_FULL);  
    }  
    list_module_def  
;  
  
list_module_def : /* no modules */  
| list_module_def module_def  
;  
  
module_def : C_MODEL C_NAME  
{  
    if (GnlCreate ($2, &CurrentGnl))  
        return (GNL_MEMORY_FULL);  
    }  
    C_INPUTS list_inputs  
    C_OUTPUTS list_outputs  
{  
    if ((GnlStatus = GnlCreateAllSignals (CurrentGnl)))  
        return (GnlStatus);  
    }  
    body  
    C_END  
;  
  
list_inputs :  
| list_inputs1
```

```

;

list_outputs      : 
|   list_outputs1
;

list_inputs1      : C_NAME
{
    /* We create a Var corresponding to the port. */
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
          (CurrentGnl, $1, &Var)))
    {
        if (GnlStatus == GNL_VAR_EXISTS)
            return (GNL_VAR_EXISTS);

        return (GNL_MEMORY_FULL);
    }
    SetGnlVarLineNumber (Var, yylineno);
    SetGnlVarDir (Var, GNL_VAR_INPUT);
}

|   list_inputs1 C_NAME
{
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
          (CurrentGnl, $2, &Var)))
    {
        if (GnlStatus == GNL_VAR_EXISTS)
            return (GNL_VAR_EXISTS);

        return (GNL_MEMORY_FULL);
    }
    SetGnlVarLineNumber (Var, yylineno);
    SetGnlVarDir (Var, GNL_VAR_INPUT);
}

;

list_outputs1      : C_NAME
{
    /* We create a Var corresponding to the port. */
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
          (CurrentGnl, $1, &Var)))
    {
        if (GnlStatus == GNL_VAR_EXISTS)
            return (GNL_VAR_EXISTS);

        return (GNL_MEMORY_FULL);
    }
    SetGnlVarLineNumber (Var, yylineno);
    SetGnlVarDir (Var, GNL_VAR_OUTPUT);
}

|   list_outputs1 C_NAME
{
    if ((GnlStatus = GnlVarCreateAndAddInHashTable
          (CurrentGnl, $2, &Var)))
    {

```

cparse.y

```
        if (GnlStatus == GNL_VAR_EXISTS)
            return (GNL_VAR_EXISTS);

        return (GNL_MEMORY_FULL);
    }
    SetGnlVarLineNumber (Var, yylineno);
    SetGnlVarDir (Var, GNL_VAR_OUTPUT);
}
;

body : list_assigns_instances
;

list_assigns_instances :
| list_assigns_instances assign_or_instance
;

assign_or_instance : instance
;

instance : C_GATE gate
{
    if (BListCreate (&VarListSignal))
        return (GNL_MEMORY_FULL);
}
list_signals
{
    if (GnlAnalyzeInstance (CurrentGnl, TheGate,
                           NULL,
                           NULL, VarListSignal,
                           yylineno, 1))
        return (GNL_MEMORY_FULL);
}
;

gate : C_NOT
{
    TheGate = 263 /* G_NOT in tokens.h */;
}
| C_NOR
{
    TheGate = 262; /* G_NOR in tokens.h */
}
| C_OR
{
    TheGate = 260; /* G_OR in tokens.h */
}
| C_AND
{
    TheGate = 259; /* G_AND in tokens.h */
}
| C_NAND
{
    TheGate = 261; /* G_NAND in tokens.h */
}
| C_XOR
{
    TheGate = 266; /* G_XOR in tokens.h */
}
| C_XNOR
{
    TheGate = 267; /* G_XNOR in tokens.h */
}
| C_ZERO
{
    TheGate = 0; /* G_NOR in tokens.h */
}
;

list_signals :
| list_signals1
;
```

```

list_signals1      : C_NAME '=' C_NAME
{
    if (GnlGetVarFromName (CurrentGnl, $3, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
    {
        if ((GnlStatus = GnlVarCreateAndAddInHashTable
              (CurrentGnl, $3, &Var)))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals (CurrentGnl), Var))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (CurrentGnl,
                        GnlNbLocal (CurrentGnl)+1);
    }
    else
        return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (VarListSignal, (int)Var))
        return (GNL_MEMORY_FULL);
}
| list_signals1  C_NAME '=' C_NAME
{
    if (GnlGetVarFromName (CurrentGnl, $4, &Var))
    {
        if (GNL_VAR_NOT_EXISTS)
    {
        if ((GnlStatus = GnlVarCreateAndAddInHashTable
              (CurrentGnl, $4, &Var)))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (GnlLocals (CurrentGnl), Var))
            return (GNL_MEMORY_FULL);
        SetGnlNbLocal (CurrentGnl,
                        GnlNbLocal (CurrentGnl)+1);
    }
    else
        return (GNL_MEMORY_FULL);
    }
    if (BListAddElt (VarListSignal, (int)Var))
        return (GNL_MEMORY_FULL);
}
;

%%

/*-----*/
/* yyerror */ *
/*-----*/
yyerror ()
{
    fprintf (stderr,"C-READER (l.%d): syntax error near '%s'\n",
            yylineno, yytext);
    exit (1);
}

```

cparse.y

```
/*-----*/
```

dmp_util.c

dmp_util.c

```
static void split_mem_block(struct MEMORY_BLOCK_REC *p)
{ int i,reduced=1,sz;
  struct MEMORY_BUFFER_REC *ptr;
  while (reduced) {
    reduced=0;
    for (i=0;i<3 && i<_mmbr_size_no;i++) {
      sz = _mmbr_size[i] ;
#ifndef DEBUGGING
      sz += 8 ;
#endif
    }
    if (p->size >= sz) {
      reduced = 1;
      ptr = (MEMORY_BUFFER_REC *) p->ptr;
      ptr->next = _mmbr_ptr[i];
      _mmbr_ptr[i] = ptr;
      p->size -= sz;
      p->ptr += sz;
    }
  }
}
void *dmp_malloc(int n)
{ int sz;
  void *ptr;
#ifndef DMP_OFF && DEBUGGING
  ptr = (char *) malloc(n) ;
  memset((char *) ptr,0,n);
#else
  register struct MEMORY_BLOCK_REC *amb,*prev=NULL;
#ifndef DEBUGGING
  assert((n%4)==0);
  sz = n + 8 ;
#else
  sz = n ;
#endif
  if (avl_mblk==NULL) {
    avl_mblk = amb = (struct MEMORY_BLOCK_REC *) malloc(sizeof(struct
MEMORY_BLOCK_REC));
    amb->ptr = (char *) malloc(MEMORY_BLOCK_SIZE);
#ifndef DEBUGGING
    mem_size_used += MEMORY_BLOCK_SIZE;
#endif
    if (amb->ptr==NULL) malloc_fail();
    memset((char *) amb->ptr,0,MEMORY_BLOCK_SIZE);
    amb->size= MEMORY_BLOCK_SIZE;
    amb->next= NULL;
  } else {
    for (amb=avl_mblk;amb!=NULL; prev=amb,amb=amb->next)
      if (amb->size>=sz) break;
    if (amb==NULL) {
      prev->next= amb=(struct MEMORY_BLOCK_REC *) malloc(sizeof(struct
MEMORY_BLOCK_REC));
      amb->ptr = (char *) malloc(MEMORY_BLOCK_SIZE);
#ifndef DEBUGGING
      mem_size_used += MEMORY_BLOCK_SIZE;
#endif
      if (amb->ptr==NULL) malloc_fail();
    }
  }
#endif
}
```

dmp_util.c

```
        memset((char *) amb->ptr,0,MEMORY_BLOCK_SIZE);
        amb->size = MEMORY_BLOCK_SIZE;
        amb->next= NULL;
    }
}
amb->size -= sz;
ptr = (void *) amb->ptr;
amb->ptr += sz;
if (amb->size<32) {
    split_mem_block(amb);
    if (prev==NULL) avl_mblk = amb->next;
    else prev->next = amb->next;
    amb->next = used_mblk;
    used_mblk = amb;
}
#endif
return(ptr);
}
/*-----*/
#if (DEBUGGING)
#define DETECT 1
#define ANALYSIS 2
#define LOCATE 3
#define MALLOC_STREAM "/tmp/_malloc_stream_"
#define FREE_STREAM "/tmp/_free_stream_"
#define MEMLEAK_STREAM "/tmp/_mem_leak_"
#define FREEERR_STREAM "/tmp/_free_err_"
#define DETECT_BUF_SIZE 64
#define MEM_ANY_SIZE 64
#define LEAKFLAG '?'
#define FERRFLAG '!'
#define OK_FLAG '.'
static int memleak_detecting=0,memleak_locating=0;
static int memleak_hold_chking=0;
static int maddr_no=0,faddr_no=0;
static void *maddr_buf [DETECT_BUF_SIZE];
static void *faddr_buf [DETECT_BUF_SIZE];
static FILE *mfd=NULL,*ffd=NULL;
static FILE *mafd=NULL,*fafd=NULL;
static unsigned int no_of_maddr=0,no_of_faddr=0;
static char maddr_flag[DETECT_BUF_SIZE],faddr_flag[DETECT_BUF_SIZE];
static int maddir=DETECT_BUF_SIZE,faddir=DETECT_BUF_SIZE;
struct MEMORY_ANY_REC
{ struct MEMORY_ANY_REC *next;
  void **addr;
  char *flag;
  int match,no; } ;
typedef struct MEMORY_ANY_REC MEMORY_ANY_REC;
static struct MEMORY_ANY_REC *mem_any_list=NULL;

static void dump_maddr(void *ptr)
{ if (ptr!=NULL) {
    maddr_buf[maddr_no++] = ptr;
    no_of_maddr++;
```

dmp_util.c

```
}

if (maddr_no>0)
if ((ptr==NULL) || (maddr_no == DETECT_BUF_SIZE)) {
    (void) fwrite(maddr_buf,sizeof(void *),maddr_no,mfd);
    maddr_no = 0;
}
}

static void dump_faddr(void *ptr)
{ if (ptr!=NULL) {
    faddr_buf[faddr_no++] = ptr;
    no_of_faddr++;
}
if (faddr_no>0)
if ((ptr==NULL) || (faddr_no == DETECT_BUF_SIZE)) {
    (void) fwrite(faddr_buf,sizeof(void *),faddr_no,ffd);
    faddr_no = 0;
}
}

static struct MEMORY_ANY_REC *get_mem_any_rec(void)
{ void *addr[MEM_ANY_SIZE];
    struct MEMORY_ANY_REC *ptr;
    int i,no;
    no = fread(addr,sizeof(void *),MEM_ANY_SIZE,mfd) ;
    if (no==0) return(NULL);
    ptr = (struct MEMORY_ANY_REC *) malloc(sizeof(struct MEMORY_ANY_REC));
    ptr->match = 0;
    ptr->next = NULL;
    ptr->no = no;
    ptr->flag = (char *) malloc(sizeof(char)*MEM_ANY_SIZE);
    ptr->addr = (void *) malloc(sizeof(void *)*MEM_ANY_SIZE);
    for (i=0;i<no;i++) {
        ptr->flag[i]=LEAKFLAG;
        ptr->addr[i]=addr[i];
    }
    return(ptr);
}

static void open_memany_stream(int type)
{
    if (type == DETECT) {
        mfd = fopen(MALLOC_STREAM, "w");
        ffd = fopen(FREE_STREAM, "w");
        memleak_detecting = 1;
        memleak_locating = 0;
        assert(mfd!=NULL);
        assert(ffd!=NULL);
    } else if (type == ANALYSIS) {
        memleak_detecting = 0;
        memleak_locating = 0;
        mfd = fopen(MALLOC_STREAM, "r");
        ffd = fopen(FREE_STREAM, "r");
        mafd = fopen(MEMLEAK_STREAM, "w");
        fafd = fopen(FREEERR_STREAM, "w");
        fprintf(stderr,"<<<<  MEMORY LEAKING ANALYZING >>>> please be patient
...\\n");
        assert(mfd!=NULL);
    }
}
```

dmp_util.c

```
assert(ffd!=NULL);
assert(mafd!=NULL);
assert(fafd!=NULL);
} else if (type == LOCATE) {
    mafd = fopen(MEMLEAK_STREAM, "r");
    fffd = fopen(FREEERR_STREAM, "r");
    fprintf(stderr, "<<< MEMORY LEAKING LOCATING >>>\n");
    memleak_detecting = 0;
    memleak_locating = 1;
    assert(mafd!=NULL);
    assert(fafd!=NULL);
} else { fprintf(stderr, " open_memany_stream type error\n"); }
}
static void close_memany_stream(int type)
{
    assert(type==DETECT || type==LOCATE || type==ANALYSIS);
    if (type==DETECT) {
        assert(mfd!=NULL && ffd!=NULL);
        dump_maddr(NULL);
        dump_faddr(NULL);
        fprintf(stderr, "[] # of mem alloc(free) =
%d(%d)\n", no_of_maddr, no_of_faddr);
    }
    if (mfd!=NULL) { fclose(mfd); mfd=NULL; }
    if (ffd!=NULL) { fclose(ffd); ffd=NULL; }
    if (mafd!=NULL) { fclose(mafd); mafd=NULL; }
    if (fafd!=NULL) { fclose(fafd); fafd=NULL; }
}
void memleak_analysis_off(void) { memleak_hold_chking=1; }
void memleak_analysis_on(void) { memleak_hold_chking=0; }
void memleak_analysis_init(char c)
{ if (c=='T') open_memany_stream(DETECT);
  else if (c=='C') open_memany_stream(LOCATE);
  memleak_hold_chking = 0;
}

void memleak_analysis(char c)
{ int i,j,k,free_no,mleak=0,ferr=0;
  void *faddr[MEM_ANY_SIZE];
  char flag[MEM_ANY_SIZE];
  struct MEMORY_ANY_REC *head;
  if (c=='C') close_memany_stream(LOCATE);
  if (c!='T') return;
  (void) close_memany_stream(DETECT);
  (void) open_memany_stream(ANALYSIS);
  while (1) {
    free_no = fread(faddr,sizeof(void *),MEM_ANY_SIZE,ffd) ;
    for (i=0;i<free_no;i++) flag[i] = FERRFLAG;

    for (i=0;i<free_no;i++) {
      if (mem_any_list==NULL) mem_any_list = get_mem_any_rec();
      head = mem_any_list;
      while (head!=NULL) {
        for (j=0;j<head->no;j++)
        if (!(head->flag[j]==OK_FLAG))
        if (head->addr[j]==faddr[i]) {
          flag[i]=head->flag[j]=OK_FLAG;
```

dmp_util.c

```
    head->match++;
    break;
}
if (j<head->no) break;
if (head->next==NULL) {
    head->next = get_mem_any_rec();
}
head=head->next;
if (head==NULL) { /* EOF */
    flag[i] = FERRFLAG;
    break;
}
}
}
while (mem_any_list!=NULL) {
    if ((mem_any_list->match==MEM_ANY_SIZE) || (free_no!=MEM_ANY_SIZE)) {
        head = mem_any_list->next;
        fwrite(mem_any_list->flag,sizeof(char),mem_any_list->no,maf);
        for (k=0;k<mem_any_list->no;k++)
            if (mem_any_list->flag[k]==LEAKFLAG) mleak++;
        free(mem_any_list->addr);
        free(mem_any_list->flag);
        free(mem_any_list);
        mem_any_list = head;
    }
    else break;
}
if (free_no>0) {
    fwrite(flag,sizeof(char),free_no,faf);
    for (k=0;k<free_no;k++)
        if (flag[k]==FERRFLAG) ferr++;
}
if (free_no<MEM_ANY_SIZE) { /* DONE! */
    if (mleak) fprintf(stderr,"!!!! WARNING !!! %d possible memory leak
detected\n",mleak);
    else if (free_no==0) fprintf(stderr,"WARNING! ALL memory allocated are
not freed.\n");
    else      fprintf(stderr,:^) GOOD NEWS!! NO memory leak found.\n";
    if (ferr)
        fprintf(stderr,"!!!! WARNING !!! %d possible memory free error
detected\n",ferr);
    if (max_mm_size>1024*5)
        fprintf(stderr," Max Memory Usage: %d KBytes\n",max_mm_size/1024);
    else fprintf(stderr," Max Memory Usage: %d Bytes\n",max_mm_size);
    break;
}
}
(void) close_memany_stream(ANALYSIS);
}

static void mleak_error(int kind)
{
    if (kind) fprintf(stderr," possible memory leaking detected.\n");
    else      fprintf(stderr," possible incorrect memory free detected.\n");
}

static void catch_maddr(void *ptr)
```

dmp_util.c

```
{ if (maddr_i>=DETECT_BUF_SIZE) {
    (void) fread(maddr_flag,sizeof(char),DETECT_BUF_SIZE,mafd);
    maddr_i = 0;
}
if (maddr_flag[maddr_i]==LEAKFLAG) {
    mleak_error(1);
}
}

static void catch_faddr(void *ptr)
{ if (faddr_i>=DETECT_BUF_SIZE) {
    (void) fread(faddr_flag,sizeof(char),DETECT_BUF_SIZE,fafd);
    faddr_i = 0;
}
if (faddr_flag[faddr_i]==FERRFLAG) {
    mleak_error(0);
}
}

static void * memleak_any(void *ptr)
{
    assert(ptr!=NULL);
    if (memleak_hold_chking) return;
    if (memleak_detecting) dump_maddr(ptr);
    if (memleak_locating ) catch_maddr(ptr);
}

static void * ferr_any(void *ptr)
{
    assert(ptr!=NULL);
    if (memleak_hold_chking) return;
    if (memleak_detecting) dump_faddr(ptr);
    if (memleak_locating ) catch_faddr(ptr);
}

#endif

#if DEBUGGING
void dmp_error(char *s)
{
    fprintf(stderr,"dmp_error():: %s\n",s); exit(1);
}

void *dmp_set_bound(void *ptr,int i)
{ int *I;
    I = (int *) ptr;
    I[0] = I[_mmb_size[i]/sizeof(int)+1] = i;
    ptr = (void *) &(I[1]);
    return(ptr);
}

void *dmp_chk_bound(void *ptr,int i)
{ int *I;
    I = (int *) ptr;
    I = (int *) &(I[-1]);
    if (I[0]!=i) dmp_error("free_mb_ptr underflow");
    if (I[_mmb_size[i]/sizeof(int)+1]!=i) dmp_error("free_mb_ptr underflow");
    ptr = (void *) I;
}
```

dmp_util.c

```
    memset((char *) ptr,0x99,_mmb_size[i]+8);
    return(ptr);
}
#endif

void * get_mb_ptr(int i)
{ void *ptr;
#ifndef DMP_OFF && DEBUGGING
    ptr = (char *) malloc(_mmb_size[i]) ;
    memset((char *) ptr,0,_mmb_size[i]);
#else
    if (_mmb_ptr[i] != NULL) {
        ptr = (void *) _mmb_ptr[i];
        _mmb_ptr[i] = _mmb_ptr[i]->next ;
#endif DEBUGGING
    memset((char *) ptr,0,_mmb_size[i]+8);
#else
    memset((char *) ptr,0,_mmb_size[i]);
#endif
}
else {
#ifndef (DEBUGGING>1)
    _mmb_cnt[i]++;
#endif
    ptr = (void *) dmp_malloc(_mmb_size[i]);
#ifndef (DEBUGGING)
    max_mm_size += _mmb_size[i];
#endif
}
#endif DEBUGGING
    memleak_any(ptr);
    ptr = dmp_set_bound(ptr,i);
#endif
#endif
    return(ptr);
}

#ifndef (DEBUGGING)
void check_mb_ptr(void *p,int i)
{ register struct MEMORY_BUFFER_REC *mb;
#ifndef (DEBUGGING>10)
for (i=0;i<_mmb_size_no;i++)
#endif
for (mb = _mmb_ptr[i]; mb!=NULL; mb=mb->next)
if (mb == p) {
    dmp_error("fatal error: free a pointer more than once ...");
}
}
#endif
void free_mb_ptr(void *p,int i)
{ register struct MEMORY_BUFFER_REC *mb;

#ifndef DMP_OFF && DEBUGGING
    free(p);
#else
#ifndef (DEBUGGING)
    p = dmp_chk_bound(p,i);

```

dmp_util.c

```
ferr_any(p);
#endif
#if (DEBUGGING>2)
    check_mb_ptr(p,i);
#endif

mb = (MEMORY_BUFFER_REC *) p;
mb->next = _mmb_ptr[i];
_mmb_ptr[i] = mb;
#endif
}

/* =====
   char text_buffer utility
=====
void * mm_dyn_buffer(int elem_no,int elem_size,void *ptr,int ALLOC)
{ int n,*i;
  register int l,h,mid,use;

#if DMP_OFF && DEBUGGING
  n = (elem_size*elem_no+3)/sizeof(int)*sizeof(int) + sizeof(int);
  if (ALLOC)
    { i = (int *) malloc(n);
      memset((char *) i,0,n);
      i[0] = elem_no ;
      ptr = (void *) (&i[1]);
      return(ptr);
    } else if (ptr!=NULL)  {
      i = (int *) ptr;
      ptr = (&i[-1]);
      free(ptr); return(NULL);
    } else return(NULL);
#else
#if DEBUGGING
  int pos1;
  if (!ALLOC) /* 0 for free */
  { if (ptr == NULL) return(NULL);
    ferr_any(ptr);
    i = (int *) ptr;
    elem_no = i[-1];
  }
  n = (elem_size*elem_no+3)/sizeof(int)*sizeof(int) + sizeof(int) * 2;
  if (!ALLOC) /* 0 for free */
  if ( n > _mmb_size[_mmb_size_no-1] ) {
    ptr = (&i[-2]);
    i = (int *) ptr;
    if (i[0]!=elem_no || i[1]!=elem_no) /* p[0]=p[1]=64; */
      dmp_error("non mm_dyn_buffer free detected!");
  } else {
    ptr = (&i[-1]);
    i = (int *) ptr;
    pos1 = 1+(elem_no*elem_size+3)/sizeof(int);
    if (i[pos1]!=elem_no) /* p[0]=p[65]=64; */
      dmp_error("mm_dyn_buffer ary may be overflow!");
  }
}
#endif
#endif
```

dmp_util.c

```
if (!ALLOC) /* 0 for free */
{ if (ptr == NULL) return(NULL);
  i = (int *) ptr;
  elem_no = i[-1];
  ptr = (&i[-1]);
}
n = (elem_size*elem_no+3)/sizeof(int)*sizeof(int) + sizeof(int) ;
#endif

if ( n > _mmmb_size[_mmmb_size_no-1] )
{
  if (ALLOC)
    { i = (int *) malloc(n);
#ifndef DEBUGGING
    max_mm_size += n;
    mem_size_used += n;
#endif
  if (i==NULL) malloc_fail();
  memset((char *) i,0,n);
  i[0] = elem_no;
#ifndef DEBUGGING
    /* for double check! */
    i[1] = elem_no ;
    ptr = (void *) (&i[2]);
#else
    ptr = (void *) (&i[1]);
#endif
#ifndef DEBUGGING
  memleak_any(ptr);
#endif
  return(ptr);
  }
  else {
#ifndef DEBUGGING>1
    memset((char *) ptr,0x99,n);
#endif
    free(ptr); return(NULL); /* will be free all together! */
  }
}

use = l = 0; h = _mmmb_size_no-1;
while (l<=h)
{ mid = (l+h)/2;
  if (_mmmb_size[mid]<n)
    use = l = mid+1 ;
  else if (_mmmb_size[mid]>n)
  { use = mid;
    h = mid-1;
  }
  else { use = mid; break; }
}

if (ALLOC)
{ if (_mmmb_ptr[use] != NULL) {
  i = (int *) _mmmb_ptr[use];
  _mmmb_ptr[use] = _mmmb_ptr[use]->next ;
  memset((char *) i,0,_mmmb_size[use]);
}
}
```

dmp_util.c

```
    else {
#endif DEBUGGING
    max_mm_size += _mm_size[use];
#endif
#ifndef (DEBUGGING>1)
    _mm_cnt[use]++;
#endif
    i = (int *) dmp_malloc(_mm_size[use]);
}

i[0] = elem_no;
#ifndef DEBUGGING
    pos1 = 1+(elem_no*elem_size+3)/sizeof(int);
    i[pos1] = elem_no ;
#endif
ptr = (void *) (&i[1]);
#ifndef (DEBUGGING)
    memleak_any(ptr);
#endif
    return(ptr);
}
else
{ struct MEMORY_BUFFER_REC *mb;

#ifndef (DEBUGGING>2)
    check_mb_ptr(ptr,use);
#endif

#ifndef (DEBUGGING>1)
    memset((char *) ptr,0x99,_mm_size[use]);
#endif
mb = (MEMORY_BUFFER_REC *) ptr;
mb->next = _mm_ptr[use];
_mm_ptr[use] = mb;

    return(NULL);
}
#endif
}

char * copy_char_array(char *ary)
{ char *ary1;
    int i;
    if (ary == NULL) return(NULL);
    ary1 = get_char_array(sizeof_char_array(ary));
    for (i=0;i<sizeof_char_array(ary);i++)
    ary1[i] = ary[i];
    return(ary1);
}

char * copy_string(char *s)
{ char *s1;
    if (s == NULL) return(NULL);
    s1 = get_text_buffer(strlen(s));
    strcpy(s1,s);
}
```

dmp_util.c

```
    return(s1);
}

char *var_text_buffer(char *format,...)
{
    int l;
    va_list args;
    char *res;
    static FILE *var_fp=NULL;

    va_start(args,format);
    if(var_fp==NULL)
        var_fp = fopen("/dev/null","w");
    l = vfprintf(var_fp,format,args);
    res = get_text_buffer(l);
    vsprintf(res,format,args);
    va_end(args);
    return(res);
}

char * cat_strings(int n, ...)
{ va_list args;
    char **sary,*str,*head;
    int i,sz=0;
    va_start(args,n);
#if DEBUGGING
    assert(n<=32);
#endif
    sary = get_ptr_buffer(n);
    for (i=0;i<n;i++) {
        sary[i] = va_arg(args, char *);
        if (sary[i]!=NULL) sz+= strlen(sary[i]);
    }
    va_end(args);
    head = str = get_text_buffer(sz+1);
    for (i=0;i<n;i++) { char *c;
        if (sary[i]!=NULL)
            for (c=sary[i]; *c!='\0'; c++) {
                *str++ = *c;
            }
    }
    *str = '\0';
    free_ptr_buffer(sary);
    return(head);
}
```

dmp_util.h

```
#ifndef DMP_UTIL_DEF
#define DMP_UTIL_DEF 1
typedef char text_buffer;
typedef char char_array;
typedef int int_buffer;
typedef float float_buffer;
typedef short short_int ;
typedef short_int short_int_buffer;

#define get_ptr_buffer(n)  mm_dyn_buffer((n),sizeof(void *),(void *)NULL,1)
#define free_ptr_buffer(p) mm_dyn_buffer(0,sizeof(void *),(void *)p,0)

#define get_int_buffer(n)  (int *) mm_dyn_buffer((n),sizeof(int),(void *)NULL,1)

#define get_float_buffer(n) (float *) mm_dyn_buffer((n),sizeof(float),(void *)NULL,1)

#define free_int_buffer(p) mm_dyn_buffer(0,sizeof(int),(void *)p,0)
#define free_float_buffer(p) mm_dyn_buffer(0,sizeof(float),(void *)p,0)

#define get_short_int_buffer(n) (short_int *)
mm_dyn_buffer((n),sizeof(short_int),(void *)NULL,1)

#define free_short_int_buffer(p) mm_dyn_buffer(0,sizeof(short_int),(void *)p,0)

#define get_text_buffer(l)  (char *)
mm_dyn_buffer((((int)(l)+4)/sizeof(void *)),sizeof(void *),(void *)NULL,1)

#define free_text_buffer(p) mm_dyn_buffer(0,sizeof(void *),(void *)p,0)

#define get_char_array(l)  (char *) mm_dyn_buffer((l),sizeof(char),(void *)NULL,1)

#define free_char_array(p) mm_dyn_buffer(0,sizeof(char),(void *)p,0)

void *get_mb_ptr(int);
void *mm_dyn_buffer(int,int,void *,int);

#define sizeof_char_array(p) (((p)==NULL)?0:(((int*)(p))[-1]))
#define sizeof_ptr_buffer(p) (((p)==NULL)?0:(((int*)(p))[-1]))
#define sizeof_int_buffer(p) (((p)==NULL)?0:(((int*)(p))[-1]))
#define sizeof_float_buffer(p) (((p)==NULL)?0:(((int*)(p))[-1]))
#define cat_2string(s1,s2) cat_strings(2,s1,s2)
#define cat_3string(s1,s2,s3) cat_strings(3,s1,s2,s3)
#define cat_4string(s1,s2,s3,s4) cat_strings(4,s1,s2,s3,s4)
#define cat_5string(s1,s2,s3,s4,s5) cat_strings(5,s1,s2,s3,s4,s5)

char * copy_string(char *);
char * var_text_buffer(char *format,...);
```

dmp_util.h

```
char * copy_char_array(char *);
char * cat_strings(int, ...);

/* -----
   memory management routine
----- */
#endif
```

lib_tokens.h

```

#include "blist.h"
#include "gnl.h"

#define L_NAME 257
#define L_FLOAT 258
#define L_INV 259
#define L_NOTINV 260
#define L_UNKNOWN 261
#define L_GATE 262
#define L_PIN 263

typedef union {
    int          ival;
    char         *sval;
    GNL_NODE    nval;
    float        fval;
} YYSTYPE;

extern YYSTYPE yyliblval;

```

ls2min.c

```
/*-----*/  
/*-----*/  
/*      File:          ls2min.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Class: none */  
/*      Inheritance: */  
/*-----*/  
/*-----*/  
  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
#include "gnlmint.h"  
#include "gnlcube.h"  
  
/*-----*/  
extern void GnlCubeFree ();  
  
/*-----*/  
/* define */  
/*-----*/  
#define FOUND_NO_VAR      0  
#define ONLY_ONE_VAR      1  
#define MORE_THAN_ONE_VAR 2  
  
/*-----*/  
/* LsUpdateVarSet */  
/*-----*/  
void LsUpdateVarSet (VarSet, Cube, NbCells)  
    unsigned *VarSet;  
    GNL_CUBE Cube;  
    int      NbCells;  
{  
    unsigned *CubeHigh;  
    unsigned *CubeLow;  
  
    CubeHigh = GnlCubeHigh (Cube);  
    CubeLow = GnlCubeLow (Cube);  
  
    while (NbCells--)  
        *(VarSet++) |= GnlCubeVars (*CubeHigh++) , *(CubeLow++));  
}  
  
/*-----*/  
/* LsRemoveSpecifiedVar */  
/*-----*/  
int LsRemoveSpecifiedVar (VarSet, VarMask, CellNb, NbCells)  
    unsigned      *VarSet;  
    unsigned      VarMask;  
    int          CellNb;  
    int          NbCells;
```

ls2min.c

```
{  
    if ((VarSet[CellNb] &= ~VarMask) != 0)  
        return (1);  
  
    while (NbCells-- && (*(VarSet++) == 0));  
  
    return (NbCells >= 0);  
}  
  
/*-----*/  
/* LsGetFirstVar */  
/*-----*/  
int LsGetFirstVar (VarSet, NbCells, VarMask)  
    unsigned *VarSet;  
    int NbCells;  
    unsigned *VarMask;  
{  
  
    while (1)  
    {  
        NbCells--;  
        if (NbCells >= 0)  
        {  
            if (VarSet[NbCells] != 0)  
            {  
                *VarMask = VarSet[NbCells];  
                *VarMask = (*VarMask) & ~(*VarMask - 1);  
                return (NbCells);  
            }  
        }  
        else  
            return (-1);  
    }  
}  
  
/*-----*/  
/* LsCubeReverseVariable */  
/*-----*/  
void LsCubeReverseVariable (Cube, CellNb, VarMask)  
    GNL_CUBE Cube;  
    int CellNb;  
    unsigned VarMask;  
{  
    GnlCubeHigh(Cube)[CellNb] ^= (VarMask);  
    GnlCubeLow(Cube)[CellNb] ^= (VarMask);  
}  
  
/*-----*/  
/* LsCubeRemoveVariable */  
/*-----*/  
void LsCubeRemoveVariable (Cube, CellNb, VarMask)  
    GNL_CUBE Cube;  
    int CellNb;  
    unsigned VarMask;  
}
```

ls2min.c

```
GnlCubeHigh(Cube) [CellNb] &= ~ (VarMask) ;
GnlCubeLow(Cube) [CellNb] |= (VarMask) ;
}

/*
 * LsCubeRemoveVariable
 */
int LsMonoFormVarExists (Cube, VarSet, NbCells)
    GNL_CUBE Cube;
    unsigned *VarSet;
    int          NbCells;
{
    unsigned *High;
    unsigned *Low;

    High = GnlCubeHigh(Cube);
    Low = GnlCubeLow(Cube);

    while (NbCells--)
    {
        if (GnlCubeSignificantVars (* (High++) , * (Low++) , * (VarSet++)))
            return (1);
    }

    return (0);
}

/*
 * GnlCubeCompatible
 */
/* Assumption: Size < 0 indicates that only (High,Low) should be */
/* considered. Otherwise the complete Cube has to be processed. */
int GnlCubeCompatible (Cube, High, Low, NbCells, TypeInstantiate)
    GNL_CUBE Cube;
    unsigned *High;
    unsigned *Low;
    int          NbCells;
    int          TypeInstantiate;
{
    unsigned *CubeHigh;
    unsigned *CubeLow;

    if (NbCells <= 0)
    {
        NbCells = -NbCells;
        return (!GnlCubeVarClash (*High, *Low,
                                  GnlCubeHigh(Cube) [NbCells],
                                  GnlCubeLow(Cube) [NbCells],
                                  TypeInstantiate));
    }

    for (CubeHigh = GnlCubeHigh(Cube), CubeLow = GnlCubeLow(Cube); NbCells--;
```

ls2min.c

```
(High++) , (Low++) , (CubeHigh++) , (CubeLow++) )
{
    if (GnlCubeVarClash (* (High) , * (Low) , * (CubeHigh) , * (CubeLow) ,
                           TypeInstantiate))
        return (0);
}

return (1);
}

/*-----
/* LsInstantiate
*/
/*-----*/
/* Given the (High, Low) parts of an instantiator, find the Cubes */
/* compatible with it. The instantiator uses the conventionnal Cube */
/* values: 00 for normal form variable, 11 for complement form variable */
/* and 01 for absence of variable (10 is not used) */
/*
GNL_STATUS LsInstantiate (ListMaxCoverCubes, High, Low, NbCells,
                          CompatibleList, TypeInstantiate)
{
    BLIST      ListMaxCoverCubes;
    unsigned   *High;
    unsigned   *Low;
    int         NbCells;
    BLIST      CompatibleList;
    int         TypeInstantiate;
    int         i;
    GNL_CUBE   CubeI;

    for (i=0; i<BListSize (ListMaxCoverCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListMaxCoverCubes, i);
        if (GnlCubeCompatible (CubeI, High, Low, NbCells, TypeInstantiate))
        {
            if (BListAddElt (CompatibleList, (int)CubeI))
                return (GNL_MEMORY_FULL);
        }
    }
    return (GNL_OK);
}

/*-----
/* LsRemoveCubeVars
*/
/*-----*/
int LsRemoveCubeVars (VarSet, Cube, Nbcells)
{
    unsigned   *VarSet;
    GNL_CUBE   Cube;
    int         Nbcells;
    unsigned   VarsRemain;
    unsigned   *CubeHigh;
    unsigned   *CubeLow;
```

ls2min.c

```
CubeHigh = GnlCubeHigh(Cube);
CubeLow = GnlCubeLow(Cube);
VarsRemain = 0;
while (Nbcells--)
{
    VarsRemain |= (* (VarSet++) &= GnlCubeNonVars (* (CubeHigh++),
                                                       * (CubeLow++)));
}

return (VarsRemain != 0);
}

/*-----
/* GnlCubeIsATautologyForVarSet
*/
int GnlCubeIsATautologyForVarSet (Cube, VarSet, NbCells)
    GNL_CUBE Cube;
    unsigned *VarSet;
    int          NbCells;
{
    unsigned *CubeHigh;
    unsigned *CubeLow;

    CubeHigh = GnlCubeHigh(Cube);
    CubeLow = GnlCubeLow(Cube);

    while (NbCells--)
    {
        if (GnlCubeSignificantVars (* (CubeHigh++) , * (CubeLow++) ,
                                     * (VarSet++)))
            return (0);
    }

    return (1);
}

/*-----
/* LsDetermineTautology
*/
/* This procedure returns 1 iff a "tautology" Cube exists in list
/* 'LCubes'.
*/
int LsDetermineTautology (LCubes, VarSet, NbCells)
    BLIST      LCubes;
    unsigned *VarSet;
    int          NbCells;
{
    int          i;
    GNL_CUBE CubeI;

    for (i=0; i<BListSize (LCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (LCubes, i);
        if (GnlCubeIsATautologyForVarSet (CubeI, VarSet, NbCells))
            return (1);
    }
}
```

ls2min.c

```
}

    return (0);
}

/*-----
/* LsChooseBestVar
 *-----
GNL_STATUS LsChooseBestVar (ListCubes, VarSet, NbCells, VarMaskBest,
                            CellNbBest)
{
    BLIST      ListCubes;
    unsigned   *VarSet;
    int        NbCells;
    unsigned   *VarMaskBest;
    int        *CellNbBest;
{
    int          MaxONOFFOccur;
    int          MinDifference;
    int          NbONOccur;
    int          NbOFFOccur;
    int          TotalOccur;
    unsigned   VarMask;
    unsigned   *VarSet1;
    int          CellNb;
    int          i;
    GNL_CUBE CubeI;

    *CellNbBest = 0;
    MaxONOFFOccur = 0;
    MinDifference = BListSize (ListCubes) + 1;

    if (MintCopy (VarSet, NbCells, &VarSet1))
        return (GNL_MEMORY_FULL);

    while ((CellNb = LsGetFirstVar (VarSet1, NbCells, &VarMask)) >= 0)
    {
        (void)LsRemoveSpecifiedVar (VarSet1, VarMask, CellNb, NbCells);
        NbONOccur = NbOFFOccur = 0;

        for (i=0; i<BListSize (ListCubes); i++)
        {
            CubeI = (GNL_CUBE)BListElt (ListCubes, i);
            if (GnlCubeSignificantONVars (GnlCubeHigh(CubeI) [CellNb],
                                         GnlCubeLow(CubeI) [CellNb],
                                         VarMask))
                NbONOccur++;
            else if (GnlCubeSignificantOFFVars (
                         GnlCubeHigh(CubeI) [CellNb],
                         GnlCubeLow(CubeI) [CellNb],
                         VarMask))
                NbOFFOccur++;
        }

        if (((TotalOccur = NbONOccur + NbOFFOccur) > MaxONOFFOccur) ||
            (TotalOccur == MaxONOFFOccur) &&

```

ls2min.c

```
        (IntDiff(NbONOccur, NbOFFOccur) < MinDifference)))
    {
        MaxONOFFOccur = TotalOccur;
        MinDifference = IntDiff(NbONOccur, NbOFFOccur);
        *CellNbBest = CellNb;
        *VarMaskBest = VarMask;
    }
}

free ((char*)VarSet1);

return (GNL_OK);
}

/*-----
/* GnlCubeUniVar
/*-----*/
int GnlCubeUniVar (Cube, VarSet, NbCells, UVarCubeHigh, UVarCubeLow,
                   UVarCellNumber)
{
    GNL_CUBE Cube;
    unsigned *VarSet;
    int      NbCells;
    unsigned *UVarCubeHigh;
    unsigned *UVarCubeLow;
    int      *UVarCellNumber;

    {
        unsigned *CubeHigh;
        unsigned *CubeLow;
        unsigned CubeVars;
        int      i;
        int      FoundOne_1 = 0;

        *UVarCellNumber = -1;
        CubeHigh = GnlCubeHigh(Cube);
        CubeLow = GnlCubeLow(Cube);

        for (i = 0; i < NbCells; i++, CubeHigh++, CubeLow++, VarSet++)
        {
            CubeVars = GnlCubeSignificantVars (*CubeHigh, *CubeLow, *VarSet);

            switch (MintHasMoreOrLessThanOne_1 (CubeVars)) {
                case 1:           /* More than one "1" found */
                    return (MORE_THAN_ONE_VAR);

                case 0:           /* Only one "1" found */
                    if (*UVarCellNumber >= 0)
                        return (MORE_THAN_ONE_VAR);
                    *UVarCubeHigh = *CubeHigh & *VarSet;
                    *UVarCubeLow = *CubeLow | ~*VarSet;
                    *UVarCellNumber = i;
                    FoundOne_1 = 1;
                    break;

                default:
                    /* No "1" found */
            }
        }
    }
}
```

ls2min.c

```
    }

}

return (FoundOne_1 ? ONLY_ONE_VAR : FOUND_NO_VAR);
}

/*-----
/* LsDetermineUnivar
/*-----
/* This procedure returns 1 iff a Univar Cube exists in list ListCubes */
/*-----*/
int LsDetermineUnivar (ListCubes, VarSet, NbCells, UVarHigh, UVarLow,
                      UVarCellNumber)
BLIST      ListCubes;
unsigned   *VarSet;
int        NbCells;
unsigned   *UVarHigh;
unsigned   *UVarLow;
int        *UVarCellNumber;
{
    int            i;
    GNL_CUBE CubeI;

    for (i=0; i<BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        if (ONLY_ONE_VAR == GnlCubeUniVar (CubeI, VarSet, NbCells,
                                           UVarHigh, UVarLow, UVarCellNumber))
            return (1);
    }
    return (0);
}

/*-----
/* LsDetermineVariableForms
/*-----
/* This procedure calculates a Cube indicating the forms in which the */
/* variables appear in the Cubes of the List. Convention is the */
/* following: */
/* 00 (11) : var appears only in complement (direct) form. */
/* 01 : var does not appear, 10 : var appears in both forms. */
/*-----*/
GNL_STATUS LsDetermineVariableForms (ListCubes, NbCells, VarSet, NewCube)
BLIST      ListCubes;
int        NbCells;
unsigned   *VarSet;
GNL_CUBE  *NewCube;
{
    int            i;
    GNL_CUBE CubeI;

    if (GnlCubeFullAndCreate (NbCells, NewCube))
```

ls2min.c

```
    return (GNL_MEMORY_FULL);

for (i=0; i<BListSize (ListCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (ListCubes, i);
    GnlCubeCumulativeMult (*NewCube, CubeI, NbCells);
}

while (NbCells--)
{
    GnlCubeHigh(*NewCube) [NbCells] &= VarSet [NbCells];
    GnlCubeLow(*NewCube) [NbCells] |= ~VarSet [NbCells];
}

return (GNL_OK);
}

/*-----
/* LsTautologyProof
*/
/*-----*/
/* This procedure returns 1 iff a ListCubes is a tautology for vars in */
/* VarSet. VarSet can be changed. */
*/
/*-----*/
GNL_STATUS LsTautologyProof (ListCubes, NbCells, VarSet, Proof)
    BLIST    ListCubes;
    int      NbCells;
    unsigned *VarSet;
    int      *Proof;
{
    BLIST    CurListCubes;
    BLIST    NewListCubes0;
    BLIST    NewListCubes1;
    int      CellNb;
    unsigned VarFormHigh;
    unsigned VarFormLow;
    GNL_CUBE CubeForms;
    unsigned *VarSet1;
    BLIST    ListAux;

    if (BListCreateWithSize (BListSize (ListCubes), &NewListCubes1))
        return (GNL_MEMORY_FULL);

    if ((CurListCubes = (BLIST)malloc (1, sizeof(BLIST_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    CurListCubes->Size = ListCubes->Size;
    CurListCubes->NbElt = ListCubes->NbElt;
    CurListCubes->Adress = ListCubes->Adress;

    *Proof = -1;
    while (LsDetermineUnivar (ListCubes, VarSet, NbCells, &VarFormHigh,
                               &VarFormLow, &CellNb))
    {
        LsInstantiate (CurListCubes, &VarFormHigh, &VarFormLow, -CellNb,
                      NewListCubes1, INVERSE_INSTANTIATE);
    }
}
```

```

if (BListSize (NewListCubes1) == 0)
    *Proof = 0;
else if (!LsRemoveSpecifiedVar (VarSet,
                                GnlCubeVars (VarFormHigh, VarFormLow),
                                CellNb, NbCells))
    *Proof = 3;
else if (LsDetermineTautology (NewListCubes1, VarSet, NbCells))
    *Proof = 3;

if (*Proof >= 0)
{
    BListQuickDelete (&NewListCubes1);
    free ((char *) CurListCubes);
    return (GNL_OK);
}

CurListCubes->Size = NewListCubes1->Size;
CurListCubes->NbElt = NewListCubes1->NbElt;
CurListCubes->Adress = NewListCubes1->Adress;
NewListCubes1->NbElt = 0;
}

while (1)
{
    if (LsDetermineVariableForms (CurListCubes, NbCells, VarSet,
                                  &CubeForms))
        return (GNL_MEMORY_FULL);

    if (!LsMonoFormVarExists (CubeForms, VarSet, NbCells))
        break;

    /* Test if all variables are monoforms */
    if (!LsRemoveCubeVars (VarSet, CubeForms, NbCells))
        *Proof = 0;
    else
    {
        LsInstantiate (CurListCubes, GnlCubeHigh(CubeForms),
                      GnlCubeLow(CubeForms),
                      NbCells, NewListCubes1, INVERSE_INSTANTIMATE);

        /* 3 pterms with at least two variables cannot constitute */
        /* a tautology */
        if (BListSize (NewListCubes1) < 4)
            *Proof = 0;
    }

    if (*Proof >= 0)
    {
        BListQuickDelete (&NewListCubes1);
        free ((char *) CurListCubes);

        GnlCubeFree (&CubeForms);

        return (GNL_OK);
    }
}

```

ls2min.c

```
CurListCubes->Size = NewListCubes1->Size;
CurListCubes->NbElt = NewListCubes1->NbElt;
CurListCubes->Adress = NewListCubes1->Adress;
NewListCubes1->NbElt = 0;

GnlCubeFree (&CubeForms);
}

GnlCubeFree (&CubeForms);

if (LsChooseBestVar (CurListCubes, VarSet, NbCells, &VarFormHigh,
                     &CellNb))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (BListSize (CurListCubes), &NewListCubes0))
    return (GNL_MEMORY_FULL);
/*BEWARE : CurListCubes is tied to NewListCubes1 */

/* Instantiate on Complement Form */          */
VarFormLow = ~0;                            /*--> Simulate DC variables      */
LsInstantiate (CurListCubes, &VarFormHigh, &VarFormLow, -CellNb,
NewListCubes0,
           INVERSE_INSTANTIATE);

/* Instantiate on Direct Form */             */
LsInstantiate (CurListCubes, &VarFormHigh, &VarFormLow, -CellNb,
NewListCubes1,
           DIRECT_INSTANTIATE);

(void)LsRemoveSpecifiedVar (VarSet, GnlCubeVars (VarFormHigh, VarFormLow),
                           CellNb, NbCells);

if (BListSize (NewListCubes0) > BListSize (NewListCubes1))
{
    ListAux = NewListCubes0;
    NewListCubes0 = NewListCubes1;
    NewListCubes1 = ListAux;
}

if (MintCopy (VarSet, NbCells, &VarSet1))
    return (GNL_MEMORY_FULL);

if (LsTautologyProof (NewListCubes0, NbCells, VarSet, Proof))
    return (GNL_MEMORY_FULL);

if (*Proof && LsTautologyProof (NewListCubes1, NbCells, VarSet1, Proof))
    return (GNL_MEMORY_FULL);

BListQuickDelete (&NewListCubes1);
BListQuickDelete (&NewListCubes0);
free ((char *) CurListCubes);
free ((char *) VarSet1);

return (GNL_OK);
}

/*-----*/
```

ls2min.c

```
/* LsInclusionTest */  
/*-----*/  
GNL_STATUS LsInclusionTest (ListMaxCoverCubes, Cube, NbCells, Proof)  
    BLIST    ListMaxCoverCubes;  
    GNL_CUBE Cube;  
    int      NbCells;  
    int      *Proof;  
{  
    int      i;  
    BLIST   CompatibleList;  
    unsigned *VarSet;  
  
    if (BListCreate (&CompatibleList))  
        return (GNL_MEMORY_FULL);  
  
    if (LsInstantiate (ListMaxCoverCubes, GnlCubeHigh(Cube),  
                      GnlCubeLow(Cube), NbCells, CompatibleList,  
                      DIRECT_INSTANTIMATE))  
        return (GNL_MEMORY_FULL);  
  
    if (BListSize (CompatibleList) == 0)  
    {  
        *Proof = 0;  
        BListQuickDelete (&CompatibleList);  
        return (GNL_OK);  
    }  
  
    if (MintCreateInit0 (NbCells, &VarSet))  
        return (GNL_MEMORY_FULL);  
  
    for (i=0; i<BListSize (CompatibleList); i++)  
    {  
        LsUpdateVarSet (VarSet, (GNL_CUBE)BListElt (CompatibleList, i),  
                        NbCells);  
    }  
  
    if (!LsRemoveCubeVars (VarSet, Cube, NbCells))  
    {  
        *Proof = 1;  
        free ((char*)VarSet);  
        BListQuickDelete (&CompatibleList);  
        return (GNL_OK);  
    }  
  
    if (LsDetermineTautology (CompatibleList, VarSet, NbCells))  
    {  
        *Proof = 2;  
        free ((char*)VarSet);  
        BListQuickDelete (&CompatibleList);  
        return (GNL_OK);  
    }  
  
    if (LsTautologyProof (CompatibleList, NbCells, VarSet, Proof))  
        return (GNL_MEMORY_FULL);  
  
    free ((char*)VarSet);
```

ls2min.c

```
    BListQuickDelete (&CompatibleList);

    return (GNL_OK);

}

/*-----*/
/* LsCubeExpand */
/*-----*/
/* This procedures takes a cube 'Cube' as input and tries to expand it */
/* in several directions. Set variables (the one occurring in the Cube) */
/* are checked with their reverse value in order to get a new cube. If */
/* the cube is still included in 'ListMaxCoverCubes' and then the */
/* expansion is kept otherwise we try the next set variable. */
/*-----*/
GNL_STATUS LsCubeExpand (NbVar, NbCells, Cube, ListMaxCoverCubes,
                         ExpandCube)
{
    int          NbVar;
    int          NbCells;
    GNL_CUBE    Cube;
    BLIST      ListMaxCoverCubes;
    GNL_CUBE *ExpandCube;
{
    unsigned *VarSet;
    unsigned VarMask;
    int          CellNb;
    int          Included;

    if (MintCreateInit0 (NbCells, &VarSet))
        return (GNL_MEMORY_FULL);

    if (GnlCubeCopy (NbCells, Cube, ExpandCube))
        return (GNL_MEMORY_FULL);

    LsUpdateVarSet (VarSet, Cube, NbCells);

    while ((CellNb = LsGetFirstVar (VarSet, NbCells, &VarMask)) >= 0)
    {
        (void)LsRemoveSpecifiedVar (VarSet, VarMask, CellNb, NbCells);

        /* We modify the cube by reverseing the value of the set var */
        LsCubeReverseVariable (*ExpandCube, CellNb, VarMask);

        if (LsInclusionTest (ListMaxCoverCubes, *ExpandCube, NbCells,
                            &Included))
            return (GNL_MEMORY_FULL);

        if (Included)
            LsCubeRemoveVariable (*ExpandCube, CellNb, VarMask);
        else
            LsCubeReverseVariable (*ExpandCube, CellNb, VarMask);
    }

    free ((char *) VarSet);
}
```

ls2min.c

```
    return (GNL_OK);
}

/*-----
/* LsMinListCubesPoor
/*-----*/
/* Procedure which performs the 2-level minimization on a list of cubes */
/* 'ListOnCubes' with a don't care set expressed by 'ListDCCubes'. */
/*-----*/
GNL_STATUS LsMinListCubesPoor (ListOnCubes, ListDCCubes, NbVar,
                               MinimizedListCubes)
{
    BLIST      ListOnCubes;
    BLIST      ListDCCubes;
    int        NbVar;
    BLIST      *MinimizedListCubes;
{
    int          i;
    int          j;
    int          NbCells;
    GNL_CUBE   CubeI;
    GNL_CUBE   CubeJ;
    BLIST      ListMaxCoverCubes;
    GNL_CUBE   ExpandCube;
    GNL_CUBE   NewCube;

    NbCells = NbOfCells (NbVar, BITS_PER_INT);

    /* The Max cover is the Union of theOn set and DC set */
    if (BListCreate (&ListMaxCoverCubes))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListOnCubes); i++)
    {
        if (BListAddElt (ListMaxCoverCubes, BListElt (ListOnCubes, i)))
            return (GNL_MEMORY_FULL);
    }

    for (i=0; i<BListSize (ListDCCubes); i++)
    {
        if (BListAddElt (ListMaxCoverCubes, BListElt (ListDCCubes, i)))
            return (GNL_MEMORY_FULL);
    }

    /* we copy 'ListOnCubes' into 'ListCubes' */
    if (BListCreate (MinimizedListCubes))
        return (GNL_MEMORY_FULL);
    for (i=0; i<BListSize (ListOnCubes); i++)
    {
        CubeI = (GNL_CUBE) BListElt (ListOnCubes, i);
        if (GnlCubeCopy (NbCells, CubeI, &NewCube))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (*MinimizedListCubes, (int) NewCube))
            return (GNL_MEMORY_FULL);
    }
}
```

ls2min.c

```
for (i=0; i<BListSize (*MinimizedListCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (*MinimizedListCubes, i);

#ifndef BUG
    /* We try to Expand the CubeI. The expanded cube is now           */
    /* 'ExpandCube'.                                                 */
    if (LsCubeExpand (NbVar, NbCells, CubeI, ListMaxCoverCubes,
                      &ExpandCube))
        return (GNL_MEMORY_FULL);

    /* We replace the old cube 'CubeI' by its expansion           */
    GnlCubeFree (&CubeI);
    BListElt (*MinimizedListCubes, i) = (int)CubeI = (int)ExpandCube;
#endif

for (j=i+1; j<BListSize (*MinimizedListCubes); j++)
{
    CubeJ = (GNL_CUBE)BListElt (*MinimizedListCubes, j);
    if (GnlCubeIncluded (NbCells, CubeJ, CubeI))
    {
        (void)BListDelInsert (*MinimizedListCubes, j+1);
        GnlCubeFree (&CubeJ);
        j--;
    }
}

BListQuickDelete (&ListMaxCoverCubes);

return (GNL_OK);
}

/*-----
/* GnlSimplifyWithCube
*/
static BLIST G_ListCubeSameSize [5000];
void GnlSimplifyWithCube (NbCells, Cube, Index, MaxCubeSize)
    int          NbCells;
    GNL_CUBE   Cube;
    int          Index;
    int          MaxCubeSize;
{
    int          i;
    int          j;
    GNL_CUBE   CubeJ;
    BLIST      NewList;

for (i=Index; i <= MaxCubeSize; i++)
{
    NewList = (BLIST)G_ListCubeSameSize [i];
    for (j=0; j<BListSize (NewList); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (NewList, j);
        if (GnlCubeIncluded (NbCells, CubeJ, Cube))
        {
```

ls2min.c

```
        (void)BListDelInsert (NewList, j+1);
        GnlCubeFree (&CubeJ);
        j--;
    }
}
}

/*-----
/* GnlAddSimplifiedCube
*/
-----*/
GNL_STATUS GnlAddSimplifiedCube (NbCells, Cube, Index)
{
    int          NbCells;
    GNL_CUBE   Cube;
    int          Index;
{
    int          i;
    BLIST      NewList;
    GNL_CUBE   CubeI;

    if (!G_ListCubeSameSize[Index])
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        G_ListCubeSameSize [Index] = NewList;
        if (BListAddElt (NewList, (int)Cube))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    NewList = G_ListCubeSameSize[Index];
    for (i=0; i<BListSize (NewList); i++)
    {
        CubeI = (GNL_CUBE)BListElt (NewList, i);
        if (GnlCubeIdentical (Cube, CubeI, NbCells))
        {
            return (GNL_OK);
        }
    }

    if (BListAddElt (NewList, (int)Cube))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}

/*-----
/* GnlCubeImage
*/
-----*/
int GnlCubeImage (NbVar, Cube1, Cube2)
{
    int          NbVar;
    GNL_CUBE   Cube1;
    GNL_CUBE   Cube2;
{
    int          LitComplement;
```

ls2min.c

```
int          i;
int          Pres1;
int          Pres2;
int          Val1;
int          Val2;

LitComplement = -1;

for (i=0; i<NbVar; i++)
{
    Pres1 = ((Val1 = MintBitValue (GnlCubeHigh(Cube1), i+1)) ==
              MintBitValue (GnlCubeLow(Cube1), i+1));
    Pres2 = ((Val2 = MintBitValue (GnlCubeHigh(Cube2), i+1)) ==
              MintBitValue (GnlCubeLow(Cube2), i+1));
    if (Pres1 != Pres2)
        return (0);

    if (Pres1 && Pres2)
    {
        if (Val1 == Val2)
            continue;

        /* We already encountered a complement literal. */
        if (LitComplement != -1)
            return (0);

        LitComplement = i;
    }
}

/* case where the two cubes are exactly the same */
if (LitComplement == -1)
    return (0);

/* we modify 'Cube1' : ex: cube1 = a.b.c, cube2 = a.!b.c */
/* cube1 becomes : cube1 = a.c */
MintResetBitValue (GnlCubeHigh(Cube1), LitComplement+1);
MintSetBitValue (GnlCubeLow(Cube1), LitComplement+1);

return (1);
}

/*
 *-----*
 * LsMinListCubes
 *-----*
 * Procedure which performs a trivial 2-level minimization. the list
 * 'ListDCCubes' is not taken into account in the minimization.
 *-----*
GNL_STATUS LsMinListCubes (ListOnCubes, ListDCCubes, NbVar,
                           MinimizedListCubes)
{
    BLIST      ListOnCubes;
    BLIST      ListDCCubes;
    int       NbVar;
    BLIST      *MinimizedListCubes;
```

ls2min.c

```
int          i;
int          j;
int          k;
int          NbCells;
int          MaxCubeSize;
int          CubeSize;
BLIST      NewList;
GNL_CUBE   NewCube;
GNL_CUBE   CubeI;
GNL_CUBE   CubeJ;
GNL_CUBE   CubeK;
int          RetryNewList;
int          Added;

NbCells = NbOfCells (NbVar, BITS_PER_INT);

/* we reset the array represententing all the cubes sorted by size */
for (i=0; i<5000; i++)
{
    G_ListCubeSameSize[i] = NULL;
}

/* We copy each cube an put it inthe right place according to its */
/* number of litteral (e.g its size). */
MaxCubeSize = 0;
for (i=0; i<BListSize (ListOnCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (ListOnCubes, i);
    if (GnlCubeCopy (NbCells, CubeI, &NewCube))
        return (GNL_MEMORY_FULL);

    CubeSize = GnlCubeLitNumber (NewCube, NbCells);

    if (CubeSize >= 5000)
    {
        fprintf (stderr,
                  " ERROR: problem of internal memory allocation\n");
        exit (1);
    }

    if (CubeSize > MaxCubeSize)
        MaxCubeSize = CubeSize;

    if (!G_ListCubeSameSize [CubeSize])
    {
        if (BListCreateWithSize (1, &NewList))
            return (GNL_MEMORY_FULL);
        G_ListCubeSameSize [CubeSize] = NewList;
    }
    NewList = G_ListCubeSameSize [CubeSize];

    if (BListAddElt (NewList, (int)NewCube))
        return (GNL_MEMORY_FULL);
}

for (i=0; i <= MaxCubeSize; i++)
```

```

{
if (i == 0)
{
    /* Particular case where the one of the cube of size 0      */
    /* is a tautology.                                         */
    /* If yes we return already.                                */
    NewList = (BLIST)G_ListCubeSameSize [0];
    for (j=0; j<BListSize (NewList); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (NewList, j);
        if (GnlCubeIsTautology (CubeJ, NbCells))
        {
            if (BListCreateWithSize (1, MinimizedListCubes))
                return (GNL_MEMORY_FULL);
            if (BListAddElt (*MinimizedListCubes, (int)CubeJ))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
    }
    continue;
}

NewList = (BLIST)G_ListCubeSameSize [i];
for (j=0; j<BListSize (NewList); j++)
{
    CubeJ = (GNL_CUBE)BListElt (NewList, j);
    RetryNewList = 0;

    for (k=j+1; k<BListSize (NewList); k++)
    {
        CubeK = (GNL_CUBE)BListElt (NewList, k);

        /* 'CubeJ' and 'CubeK' have all the same literals except*/
        /* one complemented: ex: a.b.c and a.!b.c           */
        if (GnlCubeImage (NbVar, CubeJ, CubeK))
        {
            GnlCubeFree (&CubeK);
            BListDelInsert (NewList, k+1);

            /* 'CubeJ' is now a cube with one less literal. We */
            /* will add the new cube in the i-1 list and remove*/
            /* it from the present i-th list.                  */
            BListDelInsert (NewList, j+1);

            /* We scan all the bigger cubes than this one and */
            /* try to remove them.                            */
            /* We start from Index list 'i'.                 */
            GnlSimplifyWithCube (NbCells, CubeJ, i,
                                  MaxCubeSize);

            /* we add the simplified cube of size 'i-1' in the */
            /* list of index i-1.                           */
            if (GnlAddSimplifiedCube (NbCells, CubeJ, i-1))
                return (GNL_MEMORY_FULL);

            i--;
            RetryNewList = 1;
        }
    }
}

```

```

        break;
    }
}

if (RetryNewList)
    break;
}
}

if (BListCreateWithSize (1, MinimizedListCubes))
    return (GNL_MEMORY_FULL);

for (i=1; i <= MaxCubeSize; i++)
{
    NewList = (BLIST)G_ListCubeSameSize [i];
    if (!NewList)
        continue;

    for (j=0; j<BListSize (NewList); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (NewList, j);
        if (BListAddElt (*MinimizedListCubes, (int)CubeJ))
            return (GNL_MEMORY_FULL);
    }
    BListQuickDelete (&NewList);
}

for (i=0; i<BListSize (*MinimizedListCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (*MinimizedListCubes, i);

    for (j=i+1; j<BListSize (*MinimizedListCubes); j++)
    {
        CubeJ = (GNL_CUBE)BListElt (*MinimizedListCubes, j);
        if (GnlCubeIncluded (NbCells, CubeJ, CubeI))
        {
            (void)BListDelInsert (*MinimizedListCubes, j+1);
            GnlCubeFree (&CubeJ);
            j--;
        }
    }
}

/*
fprintf (stderr, " Gain = %d --> %d\n", BListSize (ListOnCubes),
         BListSize (*MinimizedListCubes));
*/
}

return (GNL_OK);
}

/*-----*/
/* GnlMakeFunctionsFlattenable */ */
/*-----*/
GNL_STATUS GnlMakeFunctionsFlattenable (Gnl, MaxCubes)
    GNL      Gnl;
    int     MaxCubes;

```

ls2min.c

```

{
    return (GNL_OK);
}

/*-----*/
/* LsMinGnl */
/*-----*/
/* When this procedure is invoking Fields 'GnlFunctionOnSetCubes' and */
/* 'GnlFunctionOnSetCubes' of each function must have been set. The */
/* 2-level minimization will try to minimize 'GnlFunctionOnSetCubes' of */
/* each function according to the don't care set 'GnlFunctionOnSetCubes' */
/* So this list will be physically modified and pointed Cubes can be */
/* physically freed. */
/* If 'RecreateNode' is 1 then the 'FunctionOnSet' nodes are removed */
/* and recreated from the 'FunctionOnSetCubes' expression. If not */
/* the 'FunctionOnSet' are preserved. */
/*-----*/
GNL_STATUS LsMinGnl (Gnl, RecreateNode)
    GNL      Gnl;
    int     RecreateNode;
{
    int          NbVar;
    int          NbCells;
    int          i;
    GNL_FUNCTION FunctionI;
    BLIST        ListOnCubes;
    BLIST        ListDCCubes;
    BLIST        MinimizedListCubes;
    GNL_VAR      VarI;
    GNL          NewGnl;
    BLIST        ListVariables;
    GNL_NODE     NewNode;
    int          Ratio;

    /* number of variables in the design. */
    NbVar = BListSize (GnlInputs (Gnl)) + BListSize (GnlOutputs (Gnl)) +
            BListSize (GnlLocals (Gnl));
    /* Number of Ints to code the variables. */
    NbCells = NbOfCells (NbVar, BITS_PER_INT);

    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
    {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);

        ListOnCubes = GnlFunctionOnSetCubes (FunctionI);
        ListDCCubes = GnlFunctionDCSetCubes (FunctionI);

#ifdef TRACE
        /* Printing the cubes ... */
        GnlListCubesPrint (stderr, NbVar,
                           GnlFunctionOnSetCubes (FunctionI));
#endif
    }
}

```

```

if (BListSize (ListOnCubes) < 100)
{
    if (LsMinListCubes (ListOnCubes, ListDCCubes, NbVar,
                        &MinimizedListCubes))
        return (GNL_MEMORY_FULL);

    /* Replacing the Old On set expression by the minimized one */
    BListDelete (&ListOnCubes, GnlCubeFree);
    GnlFunctionOnSetCubes (FunctionI) = MinimizedListCubes;
}
else      /* For big functions... */          */
{
    Ratio = GnlListCubesLitNumber (ListOnCubes, NbCells) /
            BListSize (ListOnCubes);
    if (Ratio > 10)
    {
        if (LsMinListCubes (ListOnCubes, ListDCCubes, NbVar,
                            &MinimizedListCubes))
            return (GNL_MEMORY_FULL);

        /* Replacing the Old On set expression by the minimized one*/
        BListDelete (&ListOnCubes, GnlCubeFree);
        GnlFunctionOnSetCubes (FunctionI) = MinimizedListCubes;
    }
}

if (!RecreateView)
    return (GNL_OK);

/* we delete all the nodes of 'Gnl'. */          */
GnlFreeNodesSegments (Gnl);

if (BListCreate (&ListVariables))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (GnlInputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlInputs (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}
for (i=0; i<BListSize (GnlOutputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlOutputs (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}
for (i=0; i<BListSize (GnlLocals (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlLocals (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (GnlFunctions (Gnl)); i++)
{

```

ls2min.c

```
VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
FunctionI = GnlVarFunction (VarI);

ListOnCubes = GnlFunctionOnSetCubes (FunctionI);
if (GnlGetNodeFromListCubes (Gnl, ListVariables, ListOnCubes,
                             &NewNode))
    return (GNL_MEMORY_FULL);
SetGnlFunctionOnSet (FunctionI, NewNode);
}

return (GNL_OK);
}/*-----*/
```

lsfact.c

```
/*
 *-----*/
/* File: lsfact.c */
/* Version: 1.1 */
/* Modifications: - */
/* Documentation: - */
/* Class: none */
/* Inheritance: */
/* */
/*-----*/
/*
#define TRACE_FACTORIZATION
*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnlcube.h"
#include "lsfact.h"
#include "gnlopt.h"

#include "blist.e"

/* ----- */
extern void LsGlobalDivisorFree ();
extern void GnlCubeFree ();
extern GNL_STATUS LsDivGainFunction ();
extern GNL_STATUS LsDivChoiceFunction ();
extern GNL_STATUS LsExtractGainFunction ();
extern GNL_STATUS LsExtractChoiceFunction ();

/* ----- */
/* GLOBAL VARIABLES */
/* ----- */
static LS_OPT_PANEL      G_OptPanel;
static int                 G_NbVar;
static int                 G_NbCells;

/* ----- */
#define      NB_STEP_FACTORIZATION    13
static int STEP_FACT[NB_STEP_FACTORIZATION] =
{100, 90, 80, 70, 60, 50, 40, 30, 20, 14, 8, 1, 0};

/* ----- */
/* KernelSearchForDivision */
/* ----- */
GNL_STATUS KernelSearchForDivision (ListListCubes, NbVar,
                                    KerNbMax, LastLker, LastLLker,
                                    ListEquatToConsider)
BLIST      ListListCubes;
int         NbVar;
int         KerNbMax;
```

lsfact.c

```
BLIST      *LastLker;
BLIST      *LastLLker;
BLIST      ListEquatToConsider;
{
    int          NbCubeMax;
    int          i;
    int          I;
    int          NbCubeI;
    BLIST      ListCubesI;

#ifndef TRACE_FACTORIZATION
    fprintf (stderr,
             " Factorization: Global Kernel Extraction For Division\n");
#endif

    if (BListSize (*LastLker) == 0)
    {
        NbCubeMax = 0;

        for (i = 0; i < BListSize (ListListCubes); i++)
        {
            ListCubesI = (BLIST)BListElt (ListListCubes, i);
            if (BlistSize (ListCubesI) > NbCubeMax)
                NbCubeMax = BListSize (ListCubesI);
        }
        if (BListCreateWithSize (5*NbCubeMax, LastLLker))
            return (GNL_MEMORY_FULL);
        (*LastLLker)->NbElt = 5*NbCubeMax;
    }

    /* SEARCH OF GLOBAL KERNELS
    for (i = 0; i < BListSize (ListEquatToConsider); i++) */
    {
        I = (int)BListElt (ListEquatToConsider, i);
        ListCubesI = (BLIST)BListElt (ListListCubes, I);
        NbCubeI = BListSize (ListCubesI);
        if (NbCubeI > 1)
            if (LsExtractGlobalKernels (ListCubesI, I,
                                         *LastLker, *LastLLker,
                                         KerNbMax,
                                         G_NbVar,
                                         G_NbCells))
                return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/* -----
/* LsDivAlgebraic                                */
/* ----- */
GNL_STATUS LsDivAlgebraic (ListCubes, CoKernel, Rest)
BLIST      ListCubes;
GNL_CUBE  CoKernel;
BLIST      *Rest;
```

lsfact.c

```
{  
    int             i;  
    GNL_CUBE CubeI;  
    GNL_CUBE NewCube;  
  
    if (BListCreateWithSize (1, Rest))  
        return (GNL_MEMORY_FULL);  
  
    for (i=0; i<BListSize (ListCubes); i++)  
    {  
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);  
        if (!GnlCubeIncluded (G_NbCells, CubeI,  
                               CoKernel))  
        {  
            if (GnlCubeCopy (G_NbCells, CubeI, &NewCube))  
                return (GNL_MEMORY_FULL);  
            if (BListAddElt (*Rest, (int)NewCube))  
                return (GNL_MEMORY_FULL);  
        }  
    }  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* IntMemberOfList */  
/* ----- */  
int IntMemberOfList (L, Int)  
{  
    BLIST     L;  
    int       Int;  
    {  
        int         i;  
  
        for (i = 0; i < BListSize (L); i++)  
            if ((int) ((L->Adress)[i]) == Int)  
                return (1);  
        return (0);  
    }  
  
/* ----- */  
/* LsUpDateListKernels */  
/* ----- */  
/* List 'LAllKer' is modified by this procedure. */  
/* ----- */  
GNL_STATUS LsUpDateListKernels (LAllKer, ListEq, ListRemoved)  
{  
    BLIST     LAllKer;  
    BLIST     ListEq;  
    BLIST     ListRemoved;  
    {  
        LS_GLOB_DIVISOR      BoxI;  
        int                  i;  
        int                  k;  
        int                  IndexK;  
        BLIST               ListCok;  
        LS_COKERNEL          CokK;
```

lsfact.c

```
i = 0;
while (i < BListSize (LAllKer))
{
    BoxI = (LS_GLOB_DIVISOR)BListElt (LAllKer, i);
    ListCok = LsGlobDivisorCokernels (BoxI);
    k = 0;
    while (k < BListSize (ListCok))
    {
        CokK = (LS_COKERNEL)BListElt (ListCok, k);
        IndexK = LsCokernelFunction (CokK);
        if (IntMemberOfList (ListEq, IndexK))
        {
            LsCokernelFree (&CokK);
            BListDelInsert (ListCok, k + 1);
        }
        else
            k++;
    }

    if (BListSize (ListCok) == 0)/* We delete all the cokernels */
    {
        if (BListAddElt (ListRemoved, (int)BoxI))
            return (GNL_MEMORY_FULL);
        LsGlobalDivisorFree (&BoxI);
        /* We then delete the couple I */
        BListDelInsert (LAllKer, i + 1);
    }
    else
        i++;
}

return (GNL_OK);
}

/*
 * LsExtendListCubes
 */
GNL_STATUS LsExtendListCubes (LCube, NbCelBefore, NbCelNow)
BLIST    LCube;
int      NbCelBefore;
int      NbCelNow;
{
    int      k;
    GNL_CUBE CubeK;
    GNL_CUBE ExtCube;

    for (k = 0; k < BListSize (LCube); k++)
    {
        CubeK = (GNL_CUBE)BListElt (LCube, k);
        if (GnlCubeAndExtend (NbCelBefore, NbCelNow, CubeK, &ExtCube))
            return (GNL_MEMORY_FULL);

        GnlCubeStructFree (CubeK);
        GnlCubeHigh (CubeK) = GnlCubeHigh (ExtCube);
        GnlCubeLow (CubeK) = GnlCubeLow (ExtCube);
    }
}
```

```

        free ((char *) ExtCube);
    }

    return (GNL_OK);
}

/*
 * LsExtendListListCubes
 */
GNL_STATUS LsExtendListListCubes (ListListCubes, NbVar)
    BLIST      ListListCubes;
    int         NbVar;
{
    int          i;
    int          NbCelBefore;
    int          NbCelNow;
    BLIST      SubListI;

    NbCelBefore = (unsigned) NbOfCells (NbVar - 1, BITS_PER_INT);
    NbCelNow = (unsigned) NbOfCells (NbVar, BITS_PER_INT);

    if (NbCelBefore != NbCelNow)
    {
        for (i = 0; i < BListSize (ListListCubes); i++)
        {
            SubListI = (BLIST)BListElt (ListListCubes, i);
            if (LsExtendListCubes (SubListI, NbCelBefore, NbCelNow))
                return (GNL_MEMORY_FULL);
        }
    }
    return (GNL_OK);
}

/*
 * LsExtendListKernels
 */
GNL_STATUS LsExtendListKernels (ListKer, NbVar, ExtendKernel, TypeOfStep)
    BLIST      ListKer;
    int         NbVar;
    int         ExtendKernel;
    int         TypeOfStep;
{
    int          NbCelBefore;
    int          NbCelNow;
    int          i;
    LS_GLOB_DIVISOR     Box1;
    BLIST      Kernel;
    BLIST      ListCok;
    LS_LOC_DIVISOR    Box2;
    GNL_CUBE     Cokern;
    int          k;
    GNL_CUBE     CubeK;
    LS_COKERNEL   CokK;
    GNL_CUBE     ExtCube;
}

```

```

NbCelBefore = (unsigned) NbOfCells (NbVar - 1, BITS_PER_INT);
NbCelNow = (unsigned) NbOfCells (NbVar, BITS_PER_INT);

if (NbCelBefore == NbCelNow)      /* Actually nothing to extend */      */
    return (GNL_OK);

for (i = 0; i < BListSize (ListKer); i++)
{
    if (TypeOfStep == LS_FACT_DIVISION)
    {
        Box1 = (LS_GLOB_DIVISOR)BListElt (ListKer, i);
        Kernel = LsGlobDivisorKernel(Box1);
        ListCok = LsGlobDivisorCokernels(Box1);
    }
    else
    {
        Box2 = (LS_LOC_DIVISOR)BListElt (ListKer, i);
        Kernel = LsLocDivisorKernel(Box2);
        Cokern = LsLocDivisorCokernel(Box2);
    }

    if (ExtendKernel)
    {
        /* We extend the cubes representing the kernels. */
        for (k = 0; k < BListSize (Kernel); k++)
        {
            CubeK = (GNL_CUBE)BListElt(Kernel, k);
            if (GnlCubeAndExtend (NbCelBefore, NbCelNow, CubeK,
                                  &ExtCube))
                return (GNL_MEMORY_FULL);
            GnlCubeStructFree (CubeK);
            GnlCubeHigh (CubeK) = GnlCubeHigh (ExtCube);
            GnlCubeLow (CubeK) = GnlCubeLow (ExtCube);
            free ((char *) ExtCube);
        }
    }

    /* we extend the Cubes representing the cokernels. */
    if (TypeOfStep == LS_FACT_DIVISION)
        for (k = 0; k < BListSize (ListCok); k++)
    {
        CokK = (LS_COKERNEL)BListElt (ListCok, k);
        Cokern = LsCokernelCube (CokK);
        if (GnlCubeAndExtend (NbCelBefore, NbCelNow,
                              Cokern, &ExtCube))
            return (GNL_MEMORY_FULL);
        GnlCubeStructFree (Cokern);
        GnlCubeHigh (Cokern) = GnlCubeHigh (ExtCube);
        GnlCubeLow (Cokern) = GnlCubeLow (ExtCube);
        free ((char *) ExtCube);
    }
    else
    {
        if (GnlCubeAndExtend (NbCelBefore, NbCelNow, Cokern,
                              &ExtCube))

```

```

        return (GNL_MEMORY_FULL);
    GnlCubeStructFree (Cokern);
    GnlCubeHigh (Cokern) = GnlCubeHigh (ExtCube);
    GnlCubeLow (Cokern) = GnlCubeLow (ExtCube);
    free ((char *) ExtCube);
}
}

return (GNL_OK);
}

/*
 * LsFindBestCandidate
 */
GNL_STATUS LsFindBestCandidate (Gnl, ListListCubes, ListKer, TypeOfStep,
                                Threshold, GainFunction, ChoiceFunction,
                                BestInd)
{
    GNL          Gnl;
    BLIST      ListListCubes;
    BLIST      ListKer;
    int         TypeOfStep;
    int         Threshold;
    int         (*GainFunction) ();
    int         (*ChoiceFunction) ();
    int         *BestInd;

    {
        int             i;
        LS_GLOB_DIVISOR Box1;
        int             GainOfThisKern;
        BLIST          IndKernWithSameGain;
        LS_LOC_DIVISOR LocDivisorI;
        int             MaxGain;

        MaxGain = 0;

        if (BListCreate (&IndKernWithSameGain))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (IndKernWithSameGain, -1))
            return (GNL_MEMORY_FULL);

        if (TypeOfStep == LS_FACT_DIVISION)
        {
            for (i = 0; i < BListSize (ListKer); i++)
            {
                Box1 = (LS_GLOB_DIVISOR) BListElt (ListKer, i);

                /* Invoking the GAIN FUNCTION
                if ((*GainFunction) (Gnl, ListListCubes,
                                     LsGlobDivisorKernel(Box1),
                                     LsGlobDivisorCokernels(Box1),
                                     G_NbCells,
                                     &GainOfThisKern))
                    return (GNL_MEMORY_FULL);
                */
            }
        }
    }
}

```

```

if ((GainOfThisKern > Threshold) &&
    (GainOfThisKern >= MaxGain))
{
    if (GainOfThisKern > MaxGain)
    {
        IndKernWithSameGain->NbElt = 0;
    }
    if (BListAddElt (IndKernWithSameGain, i))
        return (GNL_MEMORY_FULL);
    MaxGain = GainOfThisKern;
}
}

else /* This is the LS_FACT_EXTRACTION */
{
    for (i = 0; i < BListSize (ListKer); i++)
    {
        LocDivisorI = (LS_LOC_DIVISOR)BListElt (ListKer, i);
        GainOfThisKern = LsLocDivisorGain (LocDivisorI);

        if ((GainOfThisKern > Threshold) &&
            (GainOfThisKern >= MaxGain))
        {
            if (GainOfThisKern > MaxGain)
            {
                IndKernWithSameGain->NbElt = 0;
            }
            if (BListAddElt (IndKernWithSameGain, i))
                return (GNL_MEMORY_FULL);
            MaxGain = GainOfThisKern;
        }
    }
}

/* we choose now the best candidate */
if ((*ChoiceFunction) (Gnl, ListListCubes, ListKer, IndKernWithSameGain,
                      G_NbCells, BestInd))
    return (GNL_MEMORY_FULL);

BListQuickDelete (&IndKernWithSameGain);

return (GNL_OK);
}

/*
 * -----
 * LsFactDivisionLoop
 * -----
 * GNL_STATUS LsFactDivisionLoop (Gnl, ListListCubes, ListEqToCompute,
 *                               LAllKer, LLAllKer, Threshold, DivisionDone)
 * GNL          Gnl;
 * BLIST        ListListCubes;
 * BLIST        ListEqToCompute;
 * BLIST        *LAllKer;
 * BLIST        *LLAllKer;
 */

```

lsfact.c

```
int             Threshold;
int             *DivisionDone;
{
    int             IndexBestCandidate;
    LS_GLOB_DIVISOR   GlobDivisorI;
    BLIST           Kernel;
    BLIST           CopyList;
    LS_COKERNEL     FirstCokBox;
    BLIST           ListOfAllCoKern;
    BLIST           ListCokernInSameEquat;
    GNL_CUBE        CoKernel;
    int             IndexOfEquat;
    BLIST           ListCubesI;
    BLIST           RestI;
    int             j;
    LS_COKERNEL     CokBoxJ;
    int             NextIndexOfEquat;
    GNL_CUBE        NextCoKernel;
    GNL_CUBE        CubeJ;
    BLIST           ListRemoved;
    int             k;
    BLIST           ListJ;
    int             m;

    *DivisionDone = 0;      /* No division done at that time          */

    if (!LsOptPanelDoDiv (G_OptPanel))
        return (GNL_OK);

    /* While we found Kernels whose gain is greater than 'Threshold' we   */
    /* perform the Division.                                              */
    while (1)
    {
        /* We look for Kernels on 'ListListCubes'.                         */
        if (KernelSearchForDivision (ListListCubes,
                                     G_NbVar,
                                     LsOptPanelMaxDivKer (G_OptPanel),
                                     LAllKer, LLAllKer,
                                     ListEqToCompute))
            return (GNL_MEMORY_FULL);

        /* We look for the Best Candidate in '*LAllKer'. This function   */
        /* returns the index of the Best Candidate. This one has a gain   */
        /* greater or equal than 'Threshold'.                                */
        if (LsFindBestCandidate (Gnl, ListListCubes, *LAllKer,
                                 LS_FACT_DIVISION, Threshold,
                                 LsOptPanelDivGainFunc (G_OptPanel),
                                 LsOptPanelDivChoiceFunc (G_OptPanel),
                                 &IndexBestCandidate))
            return (GNL_MEMORY_FULL);

        if (IndexBestCandidate == -1)
        {
            BListQuickDelete (&ListEqToCompute);
            return (GNL_OK);
        }
    }
}
```

```

else
{
    *DivisionDone = 1;
    GlobDivisorI = (LS_GLOB_DIVISOR)BListElt(*LAllKer,
                                                IndexBestCandidate);
    Kernel = LsGlobDivisorKernel (GlobDivisorI);

    /* Creation of the sub-function corresponding to the kernel */
    /* 'Kernel'. We add it at the end of list 'ListListCubes'. */
    if (GnlCopyListOfCubes (Kernel, &CopyList,
                           G_NbCells))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListListCubes, (int)CopyList))
        return (GNL_MEMORY_FULL);

    /* We have to create a new variable SFi. */
    G_NbVar++;
    G_NbCells = (unsigned) NbOfCells (G_NbVar, BITS_PER_INT);

    /* We need to extend the cubes in 'ListListCubes'. */
    if (LsExtendListListCubes (ListListCubes,
                               G_NbVar))
        return (GNL_MEMORY_FULL);

    /* We need to extend the Cubes in the Kernel structures */
    if (LsExtendListKernels (*LAllKer,
                           G_NbVar, 1,
                           LS_FACT_DIVISION))
        return (GNL_MEMORY_FULL);

    /* REnaming of 'Kernel' at the Cokernels level. */
    ListEqToCompute->NbElt = 0;
    if (BListCopyNoEltCr (LsGlobDivisorCokernels (GlobDivisorI),
                          &ListOfAllCoKern))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (1, &ListCokernInSameEquat))
        return (GNL_MEMORY_FULL);

    while (BListSize (ListOfAllCoKern))
    {
        FirstCokBox = (LS_COKERNEL)
                      BListElt (ListOfAllCoKern, 0);
        CoKernel = LsCokernelCube(FirstCokBox);

        if (BListAddElt (ListCokernInSameEquat, (int)CoKernel))
            return (GNL_MEMORY_FULL);
        IndexOfEquat = LsCokernelFunction(FirstCokBox);
        if (BListAddElt (ListEqToCompute, (int)IndexOfEquat))
            return (GNL_MEMORY_FULL);

        ListCubesI = (BLIST)BListElt (ListListCubes,
                                      IndexOfEquat);
        if (LsDivAlgebraic (ListCubesI, CoKernel, &RestI))
            return (GNL_MEMORY_FULL);
        BListDelete (&ListCubesI, GnlCubeFree);

        BListDelInsert (ListOfAllCoKern, 0 + 1);
    }
}

```

```

/* we scan 'ListOfAllCoKern' because there may be      */
/* several Cokernels belonging at the equation of index*/
/* 'IndexOfEquat', Index of the 1rst processed Cokernel*/
for (j = 0; j < BListSize (ListOfAllCoKern); j++)
{
    CokBoxJ = (LS_COKERNEL)BListElt (ListOfAllCoKern, j);
    NextIndexOfEquat = LsCokernelFunction(CokBoxJ);
    if (NextIndexOfEquat == IndexOfEquat)
    {
        NextCoKernel = LsCokernelCube(CokBoxJ);
        ListCubesI = RestI;
        if (LsDivAlgebraic (ListCubesI, NextCoKernel,
                             &RestI))
            return (GNL_MEMORY_FULL);
        BListDelete (&ListCubesI, GnlCubeFree);
        if (BListAddElt (ListCokernInSameEquat,
                         (int) NextCoKernel))
            return (GNL_MEMORY_FULL);
        BListDelInsert (ListOfAllCoKern, j + 1);
        j--;
    }
}
/* Here, 'ListCokernInSameEquat' contains kernels which*/
/* belong to equation of index 'IndexOfEquat'.          */
/* we rename in equation of index 'IndexOfEquat'. Here */
/* we have 'ListCokernInSameEquat' which is the list of*/
/* the Cokernels of the equation and 'RestI' which is   */
/* the remainder of the division. We need to add one   */
/* bit at the Cubes of list 'ListCokernInSameEquat'   */
/* corresponding to the new sub-function 'SFI'. Then we*/
/* append this list with 'RestI' in order to get the   */
/* new form of equation of index 'IndexOfEquat'. */
for (j=0; j < BListSize (ListCokernInSameEquat); j++)
{
    CubeJ = (GNL_CUBE)BListElt(ListCokernInSameEquat, j);
    MintSetBitValue (GnlCubeHigh(CubeJ),
                     G_NbVar);
    MintSetBitValue (GnlCubeLow(CubeJ),
                     G_NbVar);
}
if (GnlCopyListOfCubes (ListCokernInSameEquat, &CopyList,
                       G_NbCells))
    return (GNL_MEMORY_FULL);

if (BListAppend (CopyList, &RestI))
    return (GNL_MEMORY_FULL);

ListCokernInSameEquat->NbElt = 0;

/* We modify the equation in 'ListListCubes'.           */
BListElt (ListListCubes, IndexOfEquat) = (int)CopyList;
}

BListQuickDelete (&ListOfAllCoKern);

```

```

BListQuickDelete (&ListCokernInSameEquat);

/* This procedure returns information on objects which have been */
/* deleted in list '*LAllKer'. List 'ListRemoved' contains pointer*/
/* of these objects. */
/* if (BListCreateWithSize (1, &ListRemoved))
   return (GNL_MEMORY_FULL);

if (LsUpDateListKernels (*LAllKer, ListEqToCompute, ListRemoved))
   return (GNL_MEMORY_FULL);

/* We update list 'LLAllKer' thanks to the list 'ListRemoved'. */
/* Reminder: LAllKer and LLAllKer point on same objects. */
for (k = 0; k < BListSize (ListRemoved); k++)
{
   for (j = 0; j < BListSize (*LLAllKer); j++)
   {
      ListJ = (BLIST)BListElt (*LLAllKer, j);
      if (ListJ != NULL)
      {
         for (m = 0; m < BListSize (ListJ); m++)
         {
            if (BListElt (ListJ, m) == BListElt (ListRemoved, k))
            {
               BListDelInsert (ListJ, m + 1);
               break;
            }
         }
      }
   }
}
BListQuickDelete (&ListRemoved);

/* We add to the list of equations to recompute later on the sub- */
/* function which has been created. */
/* if (BListAddElt (ListEqToCompute, (int)(BListSize (ListListCubes)-1)))
   return (GNL_MEMORY_FULL);
}

/* End of while(1) */

return (GNL_OK);
}

/* -----
/* LsFactDivision
/* -----
GNL_STATUS LsFactDivision (Gnl, ListListCubes, Threshold, DivisionDone,
                           LAllKer, LLAllKer)
{
   GNL          Gnl;
   BLIST        *ListListCubes;
   int          Threshold;
   int          *DivisionDone;
   BLIST        *LAllKer;
   BLIST        *LLAllKer;
   int          i;
   int          NbEquations;
}

```

lsfact.c

```
GNL_VAR VarI;
GNL_FUNCTION FunctionI;
BLIST ListCubesI;
BLIST ListEqToCompute;

#ifndef TRACE_FACTORIZATION
    fprintf (stderr, " FACTORIZATION: Division\n");
#endif

if (BListSize (*ListListCubes))
    NbEquations = BListSize (*ListListCubes);
else
    NbEquations = BListSize (GnlFunctions (Gnl));

if (BListCreate (&ListEqToCompute))
    return (GNL_MEMORY_FULL);

if (BListSize (*LAllKer) == 0) /* At the beginning */ {
    for (i=0; i<NbEquations; i++)
        if (BListAddElt (ListEqToCompute, (int)i))
            return (GNL_MEMORY_FULL);
}

/* True at the first call because after '*ListListCubes' represents */
/* the current state of the boolean network. */ if (BListSize (*ListListCubes) == 0) {
    for (i=0; i<BListSize (GnlFunctions (Gnl)); i++) {
        VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
        FunctionI = GnlVarFunction (VarI);
        ListCubesI = GnlFunctionOnSetCubes (FunctionI);
        if (BListAddElt (*ListListCubes, (int)ListCubesI))
            return (GNL_MEMORY_FULL);
    }
}

if (LsFactDivisionLoop (Gnl, *ListListCubes, ListEqToCompute, LAllKer,
                       LLAllKer, Threshold, DivisionDone))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*
 * -----
 * LsDeMorganSumToProd
 * -----
 * This procedure converts a sum of cubes with one single literal into
 * a product of complemented literals.
 * Ex: i1+i2+...+in into ~i1.^~i2....^in.
 * -----
 * GNL_STATUS LsDeMorganSumToProd (ListCubes, Cube)
 * BLIST ListCubes;
 * GNL_CUBE *Cube;
```

lsfact.c

```
{  
    int             i;  
    int             j;  
    int             Val;  
    GNL_CUBE CubeI;  
  
    if (GnlCubeCreate (Cube))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlCubeAndStructCreateInit (*Cube, G_NbCells))  
        return (GNL_MEMORY_FULL);  
  
    for (i = 0; i < BListSize (ListCubes); i++)  
    {  
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);  
        for (j = 0; j < G_NbVar; j++)  
        {  
            if ((Val = MintBitValue (GnlCubeHigh(CubeI), j + 1)) ==  
                MintBitValue (GnlCubeLow(CubeI), j + 1))  
            {  
                if (Val)  
                {  
                    MintResetBitValue (GnlCubeHigh(*Cube), j + 1);  
                    MintResetBitValue (GnlCubeLow(*Cube), j + 1);  
                }  
                else  
                {  
                    MintSetBitValue (GnlCubeHigh(*Cube), j + 1);  
                    MintSetBitValue (GnlCubeLow(*Cube), j + 1);  
                }  
            }  
        }  
    }  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* LsComplementeSumOfSingleVar */  
/* ----- */  
GNL_STATUS LsComplementeSumOfSingleVar (ListListCubes, ComplementDone)  
    BLIST      ListListCubes;  
    int       *ComplementDone;  
{  
    int             i;  
    int             j;  
    GNL_CUBE CubeJ;  
    int             NbOneCube;  
    int             NotEnd;  
    BLIST      ListCubesI;  
    GNL_CUBE CubeProd;  
    BLIST      ListCubeProd;  
    BLIST      ListOfOneVarListCubesI;  
    GNL_CUBE Cube;
```

```

#ifndef TRACE_FACTORIZATION
    fprintf (stderr,
             " FACTORIZATION: Complement Sum Single Var\n");
#endif

if (!LsOptPanelDoExt (G_OptPanel))
{
    *ComplementDone = 0;
    return (GNL_OK);
}

for (i=0; i<BListSize (ListListCubes); i++)
{
    ListCubesI = (BLIST)BListElt (ListListCubes, i);
    if (BListCreateWithSize (1, &ListOfOneVarListCubesI))
        return (GNL_MEMORY_FULL);

NbOneCube = 0;
NotEnd = 1;
j = 0;
while (NotEnd)
{
    if (j<BListSize (ListCubesI))
    {
        CubeJ = (GNL_CUBE)BListElt (ListCubesI, j);
        if (GnlCubeLitNumber (CubeJ,
                               G_NbCells) == 1)
        {
            NbOneCube++;
            if (BListAddElt (ListOfOneVarListCubesI, (int)CubeJ))
                return (GNL_MEMORY_FULL);
            BListDelInsert (ListCubesI, j + 1);
        }
        else
            j++;
    }
    else
        NotEnd = 0;
}

/* If at least there is a sum of cubes with unique variables then we*/
/* can complement it. */
if (NbOneCube > 1)
{
    *ComplementDone = 1;

/* we compute the complement of this sum of cubes. */
    if (LsDeMorganSumToProd (ListOfOneVarListCubesI, &CubeProd))
        return (GNL_MEMORY_FULL);

/* We create a sub-function corresponding to this sum of cubes. */
    if (BListCreateWithSize (1, &ListCubeProd))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListCubeProd, (int)CubeProd))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (ListListCubes, (int>ListCubeProd))
        return (GNL_MEMORY_FULL);
}

```

```

/* We create a new sub-function . */  

G_NbVar++;  

G_NbCells = (unsigned) NbOfCells (G_NbVar, BITS_PER_INT);  

/* We need to extend the cubes */  

if (LsExtendListListCubes (ListListCubes,  

    G_NbVar))  

    return (GNL_MEMORY_FULL);  

if (GnlCubeCreate (&Cube))  

    return (GNL_MEMORY_FULL);  

if (GnlCubeAndStructCreateInit (Cube, G_NbCells))  

    return (GNL_MEMORY_FULL);  

MintResetBitValue (GnlCubeHigh(Cube), G_NbVar);  

MintResetBitValue (GnlCubeLow(Cube), G_NbVar);  

if (BListAddElt (ListCubesI, (int)Cube))  

    return (GNL_MEMORY_FULL);  

    BListDelete (&ListOfOneVarListCubesI, GnlCubeFree);  

}  

else  

{  

    if (NbOneCube == 1)  

    {  

        if (BListAddElt (ListCubesI,  

            BListElt (ListOfOneVarListCubesI, 0)))  

            return (GNL_MEMORY_FULL);  

    }  

    BListQuickDelete (&ListOfOneVarListCubesI);  

}  

}  

return (GNL_OK);
}  

/* ----- */  

/* KernelSearchForExtraction */  

/* ----- */  

GNL_STATUS KernelSearchForExtraction (ListCubes, NbVar, KerNbMax,  

                                         LastListker, GainFunction, Gnl,  

                                         ListListCubes)  

BLIST      ListCubes;  

int        NbVar;  

int        KerNbMax;  

BLIST      *LastListker;  

GNL_STATUS    (*GainFunction) ();  

GNL        Gnl;  

BLIST      ListListCubes;  

{  

#ifndef TRACE_FACTORIZATION
    fprintf (stderr,
            " FACTORIZATION: Local Kernel Extraction For Extraction\n");
#endif
}

```

lsfact.c

```
    return (LsExtractLocalKernels (ListCubes, *LastListker, KerNbMax,
                                   NbVar,
                                   NbofCells (NbVar, BITS_PER_INT),
                                   GainFunction, Gnl, ListListCubes));
}

/* -----
/* LsSubstituteInPTerm
/* -----
void LsSubstituteInPTerm (Cube1, Cube2, Res)
{
    GNL_CUBE Cube1;
    GNL_CUBE Cube2;
    int          *Res;
{
    int          i;

    *Res = 0;
    if (GnlCubeIncluded (G_NbCells, Cube1, Cube2))
    {
        for (i = 0; i < G_NbVar - 1; i++)
        {
            if (MintBitValue (GnlCubeHigh(Cube2), i + 1) ==
                MintBitValue (GnlCubeLow(Cube2), i + 1))
            {
                /* we set it to Phi.                      */
                MintResetBitValue (GnlCubeHigh(Cube1), i + 1);
                MintSetBitValue (GnlCubeLow(Cube1), i + 1);
            }
        }

        MintSetBitValue (GnlCubeHigh(Cube1), G_NbVar);
        MintSetBitValue (GnlCubeLow(Cube1), G_NbVar);
        *Res = 1;
    }
}

/* -----
/* LsUpDateListKerExtraction
/* -----
GNL_STATUS LsUpDateListKerExtraction (ListKern, Index, GainFunction, Gnl,
                                         ListListCubes)
{
    BLIST      ListKern;
    int         Index;
    GNL_STATUS (*GainFunction) ();
    GNL         Gnl;
    BLIST      ListListCubes;
{
    int          i;
    LS_LOC_DIVISOR BoxI;
    GNL_CUBE     TheCommonPTerm;
    LS_LOC_DIVISOR CurrentBox;
    GNL_CUBE     CoK;
    BLIST       Kernel;
    int          j;
    int          k;
    GNL_CUBE     CubeK;
    int          ComputeNewGain;
```

```

int             Res;
int             Intersect;
int             Gain;

ComputeNewGain = 0;

BoxI = (LS_LOC_DIVISOR)BListElt (ListKern, Index);
TheCommonPTerm = LsLocDivisorCokernel (BoxI);

BListDelInsert (ListKern, Index + 1);

i = 0;
while (i < BListSize (ListKern))
{
    CurrentBox = (LS_LOC_DIVISOR)BListElt (ListKern, i);
    CoK = LsLocDivisorCokernel (CurrentBox);
    Kernel = LsLocDivisorKernel (CurrentBox);

/* 1rst case: the cube common part is totally included in */
/* Cokernel. Ex: TheCommonPTerm = ab and candidate is (abcd) */
/* [ef+g+1] then it becomes: (cdSF1) [ef+g+1] */
    LsSubstituteInPTerm (CoK, TheCommonPTerm, &Res);

/* if not the case ... */
    if (!Res)
    {
        Intersect = 0;
        for (j = 0; j < G_NbVar; j++)
        {
            if ((MintBitValue (GnlCubeHigh(CoK), j + 1) ==
                  MintBitValue (GnlCubeHigh(TheCommonPTerm), j + 1)) &&
                (MintBitValue (GnlCubeLow(CoK), j + 1) ==
                  MintBitValue (GnlCubeLow(TheCommonPTerm), j + 1)))
            {
                Intersect = 1;
                break;
            }
        }

        if (Intersect)
        {
            /* If there is an intersection then 'TheCommonPTerm' may*/
            /* overlap the Cube (CoK.CubeK) where 'CubeK' is one of */
            /* the cubes of 'Kernel'. Then we have to remove this */
            /* cube 'CubeK' */
            k = 0;
            while (k < BListSize (Kernel))
            {
                CubeK = (GNL_CUBE)BListElt (Kernel, k);
                if (GnlCubeIncluded (G_NbCells,
                                      CubeK,
                                      TheCommonPTerm))
                {
                    BListDelInsert (Kernel, k + 1);
                    ComputeNewGain = 1;
                }
            }
        }
    }
}

```

```

        else
            k++;
    }
}

/* Cleaning ... */
if (BListSize (Kernel) <= 1) /* if no more kernel ... */
{
    ComputeNewGain = 0;
    BListDelInsert (ListKern, i + 1);
    BListQuickDelete (&(LsLocDivisorKernel (CurrentBox)));
    GnlCubeFree (&CoK);
    free ((char*)CurrentBox);
}
else
{
    i++;
}
}

else
{
    ComputeNewGain = 1;
    i++;
}

if (ComputeNewGain)
{
    if ((*GainFunction) (Gnl, ListListCubes,
                         LsLocDivisorKernel (CurrentBox),
                         LsLocDivisorCokernel (CurrentBox),
                         G_NbCells, &Gain))
        return (GNL_MEMORY_FULL);

    SetLsLocDivisorGain (CurrentBox, Gain);
}
}

return (GNL_OK);
}

/*
 * LsFactExtractionLoop
 */
GNL_STATUS LsFactExtractionLoop (Gnl, ListCubes, ListListCubes, ListKern,
                                 Threshold, ExtractDone)
{
    GNL          Gnl;
    BLIST        *ListCubes;
    BLIST        *ListListCubes;
    BLIST        *ListKern;
    int          Threshold;
    int          *ExtractDone;
{
    int          i;
    int          j;
    int          IndexBestCandidate;
    GNL_CUBE    TheCommonPTerm;
    GNL_CUBE    NewCommonPTerm;
}

```

lsfact.c

```
GNL_CUBE      CubeJ;
BLIST         Kernel;
LS_LOC_DIVISOR BoxI;
BLIST         ListSubFct;
int          Res;

/*ExtractDone = 0;

/* While there are Cokernels with a gain greater than 'Threshold' we */
/* performs the substitution.                                         */
while (1)
{
    if (BListSize (*ListKern) == 0)
    {
        /* we call the Cokernels research on '*ListCubes'           */
        if (KernelSearchForExtraction (*ListCubes,
                                        G_NbVar,
                                        LsOptPanelMaxExtKer (G_OptPanel),
                                        ListKern,
                                        LsOptPanelExtGainFunc (G_OptPanel),
                                        Gnl,
                                        *ListListCubes))
            return (GNL_MEMORY_FULL);
    }

    /* we look for the best candidate in '**ListKern'.           */
    if (LsFindBestCandidate (Gnl, *ListListCubes, *ListKern,
                            LS_FACT_EXTRACTION, Threshold,
                            LsOptPanelExtGainFunc (G_OptPanel),
                            LsOptPanelExtChoiceFunc (G_OptPanel),
                            &IndexBestCandidate))
        return (GNL_MEMORY_FULL);

    if (IndexBestCandidate == -1)
    {
        return (GNL_OK);
    }
    else
    {
        *ExtractDone = 1;
        BoxI = (LS_LOC_DIVISOR)BListElt (*ListKern,
                                         IndexBestCandidate);
        Kernel = LsLocDivisorKernel (BoxI);
        TheCommonPTerm = LsLocDivisorCokernel (BoxI);

        /* We create a new sub-function for the cube common part */
        if (BListCreateWithSize (1, &ListSubFct))
            return (GNL_MEMORY_FULL);
        if (GnlCubeCopy (G_NbCells,
                         TheCommonPTerm, &NewCommonPTerm))
            return (GNL_MEMORY_FULL);
        if (BlistAddElt (ListSubFct, (int)NewCommonPTerm))
            return (GNL_MEMORY_FULL);
        if (BlistAddElt (*ListListCubes, (int>ListSubFct))
            return (GNL_MEMORY_FULL);
    }
}
```

```

G_NbVar++;
G_NbCells = (unsigned) NbOfCells (G_NbVar, BITS_PER_INT);

if (LsExtendListListCubes (*ListListCubes,
                           G_NbVar))
    return (GNL_MEMORY_FULL);

if (LsExtendListKernels (*ListKern,
                        G_NbVar, 0,
                        LS_FACT_EXTRACTION))
    return (GNL_MEMORY_FULL);

if (LsUpDateListKerExtraction (*ListKern, IndexBestCandidate,
                               LsOptPanelExtGainFunc (G_OptPanel), Gnl,
                               *ListListCubes))
    return (GNL_MEMORY_FULL);

for (j=0; j<BListSize (Kernel); j++)
{
    CubeJ = (GNL_CUBE)BListElt (Kernel, j);
    LsSubstituteInPTerm (CubeJ, TheCommonPTerm, &Res);
}

BListQuickDelete (&(BoxI->Kernel));
GnlCubeFree (&TheCommonPTerm);
free ((char*)BoxI);
}

return (GNL_OK);
}

/*
/* LsFactExtraction
/*
*/
GNL_STATUS LsFactExtraction (Gnl, Threshold, ListCubes, ListListCubes,
                             ListKerExtract, ExtractDone)
{
    GNL          Gnl;
    int          Threshold;
    BLIST        *ListCubes;
    BLIST        *ListListCubes;
    BLIST        *ListKerExtract;
    int          *ExtractDone;
{
    int          i;
    int          NbEquations;
    BLIST        LPTermI;
    GNL_VAR     Vari;
    GNL_FUNCTION FunctionI;

#endif TRACE_FACTORIZATION
    fprintf (stderr, " FACTORIZATION: Extraction\n");
#endif

    if (!LsOptPanelDoExt (G_OptPanel))

```

lsfact.c

```
{  
    *ExtractDone = 0;  
    return (GNL_OK);  
}  
  
NbEquations = BListSize (*ListListCubes);  
  
if (BListSize (*ListCubes) == 0)  
{  
    if (BListSize (*ListListCubes) == 0)  
    {  
        for (i = 0; i < BListSize (GnlFunctions (Gnl)); i++)  
        {  
            VarI = (GNL_VAR) BListElt (GnlFunctions (Gnl), i);  
            FunctionI = GnlVarFunction (VarI);  
            LPTermI = GnlFunctionOnSetCubes (FunctionI);  
            if (BListAddElt (*ListListCubes, (int) LPTermI))  
                return (GNL_MEMORY_FULL);  
        }  
  
        NbEquations = BListSize (GnlFunctions (Gnl));  
  
        for (i = 0; i < NbEquations; i++)  
        {  
            LPTermI = (BLIST) BListElt (*ListListCubes, i);  
            if (BListAddNoEltCr (*ListCubes, LPTermI))  
                return (GNL_MEMORY_FULL);  
        }  
    }  
    else  
    {  
        for (i = 0; i < NbEquations; i++)  
        {  
            LPTermI = (BLIST) BListElt (*ListListCubes, i);  
            if (BListAddNoEltCr (*ListCubes, LPTermI))  
                return (GNL_MEMORY_FULL);  
        }  
    }  
}  
  
if (BListSize (*ListCubes) != 0)  
    if (LsFactExtractionLoop (Gnl, ListCubes, ListListCubes,  
                             ListKerExtract, Threshold, ExtractDone))  
        return (GNL_MEMORY_FULL);  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* LsFactorizationStep */  
/* ----- */  
GNL_STATUS LsFactorizationStep (Gnl, ListListCubes, ListCubes, ListKerDiv,  
                                ListListKerDiv, ListKerExtract, DivisionDone,  
                                ComplementDone, ExtractDone, ThresHold)  
{  
    GNL          Gnl;  
    BLIST      *ListListCubes;  
    BLIST      *ListCubes;
```

lsfact.c

```
BLIST    *ListKerDiv;
BLIST    *ListListKerDiv;
BLIST    *ListKerExtract;
int      *DivisionDone;
int      *ComplementDone;
int      *ExtractDone;
int      ThresHold;
{
    int          i;
    LS_LOC_DIVISOR LocDivisorI;

    /* If there was previously a COMPLEMENTATION or an EXTRACTION */
    if (*ComplementDone || *ExtractDone)
    {
        /* We need to erase everything and restart from scratch */
        if (*ListKerDiv)
            BListDelete (ListKerDiv, LsGlobalDivisorFree);
        if (*ListListKerDiv)
            BListDelete (ListListKerDiv, BListQuickDelete);
        if (BListCreate (ListKerDiv))
            return (GNL_MEMORY_FULL);
    }

    /* Algebraic Division */
    if (LsFactDivision (Gnl, ListListCubes, ThresHold, DivisionDone,
                        ListKerDiv, ListListKerDiv))
        return (GNL_MEMORY_FULL);

#define BUG
    /* If there was previously a DIVISION or an EXTRACTION */
    if (*DivisionDone || *ExtractDone)
        if (LsComplementeSumOfSingleVar (*ListListCubes, ComplementDone))
            return (GNL_MEMORY_FULL);
#endif

    /* If there was previously a DIVISION or a COMPLEMENTATION */
    if (*ComplementDone || *DivisionDone)
    {
        for (i=0; i<BListSize (*ListKerExtract); i++)
        {
            LocDivisorI = (LS_LOC_DIVISOR)BListElt (*ListKerExtract, i);
            BListQuickDelete (&LsLocDivisorKernel(LocDivisorI));
            GnlCubeFree (&LsLocDivisorCokernel(LocDivisorI));
            free ((char*)LocDivisorI);
        }
        (*ListKerExtract)->NbElt = 0;
        (*ListCubes)->NbElt = 0;
    }
    if (LsFactExtraction (Gnl, ThresHold, ListCubes, ListListCubes,
                          ListKerExtract, ExtractDone))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/* ----- */
```

lsfact.c

```
/* GnlGetNodeFromCube */  
/* ----- */  
GNL_STATUS GnlGetNodeFromCube (Gnl, ListVariables, Cube, NewNode)  
    GNL        Gnl;  
    BLIST     ListVariables;  
    GNL_CUBE  Cube;  
    GNL_NODE *NewNode;  
{  
    int          i;  
    int          Num;  
    int          NbVar;  
    int          NbCells;  
    int          Mask;  
    GNL_NODE VarNode;  
    GNL_NODE VarNot;  
    int          NbTreat;  
    BLIST     Sons;  
    GNL_VAR   Var;  
  
    NbVar = BListSize (ListVariables);  
    NbCells = NbOfCells (NbVar, BITS_PER_INT);  
  
    if (GnlCreateNode (Gnl, NULL, NewNode))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlCubeIsTautology (Cube, NbCells))  
    {  
        SetGnlNodeOp (*NewNode, GNL_CONSTANTE);  
        SetGnlNodeSons (*NewNode, (BLIST)1);  
        return (GNL_OK);  
    }  
  
    if (BListCreateWithSize (1, &Sons))  
        return (GNL_MEMORY_FULL);  
  
    SetGnlNodeOp (*NewNode, GNL_AND);  
    SetGnlNodeSons (*NewNode, Sons);  
  
    Num = 0;  
    i = 1;  
    while (NbVar)  
    {  
        NbVar -= (NbTreat = (NbVar > BITS_PER_INT) ? BITS_PER_INT : NbVar);  
        Mask = 1;  
        do {  
            if (GnlCubeHigh(Cube) [Num] & Mask)  
            {  
                if (GnlCubeLow(Cube) [Num] & Mask)  
                {  
                    if (GnlCreateNode (Gnl, NULL, &VarNode))  
                        return (GNL_MEMORY_FULL);  
                    SetGnlNodeOp (VarNode, GNL_VARIABLE);  
                    Var = (GNL_VAR)BListElt (ListVariables, i-1);  
                    SetGnlNodeSons (VarNode, (BLIST)Var);  
                    if (BListAddElt (Sons, (int)VarNode))  
                        return (GNL_MEMORY_FULL);  
                }  
            }  
        }  
    }  
}
```

```

        }
    }
    else if ( !(GnlCubeLow(Cube) [Num] & Mask) )
    {
        if (GnlCreateNode (Gnl, NULL, &VarNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeOp (VarNode, GNL_VARIABLE);
        Var = (GNL_VAR)BListElt (ListVariables, i-1);
        SetGnlNodeSons (VarNode, (BLIST)Var);
        if (GnlCreateNodeNot (Gnl, VarNode, &VarNot))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (Sons, (int)VarNot))
            return (GNL_MEMORY_FULL);
    }

    Mask <= 1;
    i++;
}

while (--NbTreat);
Num++;
}

if (BListSize (Sons) == 1)
{
    *NewNode = (GNL_NODE)BListElt (Sons, 0);
}

return (GNL_OK);
}

/*
/* GnlGetNodeFromListCubes
*/
GNL_STATUS GnlGetNodeFromListCubes (Gnl, ListVariables, ListCubes,
                                    NewNode)
{
    GNL          Gnl;
    BLIST       ListVariables;
    BLIST       ListCubes;
    GNL_NODE *NewNode;
{
    int          i;
    GNL_CUBE CubeI;
    BLIST       Sons;
    GNL_NODE NewSon;

    /* Case of Constante 0 */
    if (BListSize (ListCubes) == 0)
    {
        if (GnlCreateNode (Gnl, GNL_CONSTANTE, NewNode))
            return (GNL_MEMORY_FULL);
        SetGnlNodeSons (*NewNode, 0);
        return (GNL_OK);
    }

    if (BListSize (ListCubes) == 1)

```

lsfact.c

```
{  
    if (GnlGetNodeFromCube (Gnl, ListVariables,  
                           (GNL_CUBE)BListElt (ListCubes, 0),  
                           NewNode))  
        return (GNL_MEMORY_FULL);  
    return (GNL_OK);  
}  
  
if (GnlCreateNode (Gnl, GNL_OR, NewNode))  
    return (GNL_MEMORY_FULL);  
if (BListCreateWithSize (BListSize (ListCubes), &Sons))  
    return (GNL_MEMORY_FULL);  
SetGnlNodeSons (*NewNode, Sons);  
  
for (i=0; i<BListSize (ListCubes); i++)  
{  
    CubeI = (GNL_CUBE)BListElt (ListCubes, i);  
    if (GnlGetNodeFromCube (Gnl, ListVariables, CubeI, &NewSon))  
        return (GNL_MEMORY_FULL);  
    if (BListAddElt (Sons, (int)NewSon))  
        return (GNL_MEMORY_FULL);  
}  
  
return (GNL_OK);  
}  
  
/* ----- */  
/* LsBuildGnlFromGnlCubes */  
/* ----- */  
GNL_STATUS LsBuildGnlFromGnlCubes (OriginalGnl, Gnl, ListListCubes)  
{  
    GNL          OriginalGnl;  
    GNL          Gnl;  
    BLIST       ListListCubes;  
  
    int           NbNewFunctions;  
    int           i;  
    GNL_VAR      VariI;  
    GNL_FUNCTION  FunctionI;  
    BLIST       ListCubesI;  
    BLIST       NodesSegments;  
    GNL_NODE     NewNode;  
    GNL_VAR      NewVar;  
    GNL_FUNCTION  NewFunction;  
    BLIST       ListVariables;  
  
    NbNewFunctions = BListSize (ListListCubes) -  
                     BListSize (GnlFunctions (Gnl));  
  
    for (i=0; i<NbNewFunctions; i++)  
    {  
        if (GnlCreateUniqueVarWithNoTestGnlId (OriginalGnl, Gnl,  
                                              "$F", &NewVar))  
            return (GNL_MEMORY_FULL);  
        if (BListAddElt (GnlLocals (Gnl), (int)NewVar))  
            return (GNL_MEMORY_FULL);  
        SetGnlNbLocal (Gnl, GnlNbLocal (Gnl)+1);  
    }
```

```

if (GnlFunctionCreate (Gnl, NewVar, NULL, &NewFunction))
    return (GNL_MEMORY_FULL);

SetGnlVarFunction (NewVar, NewFunction);

if (BListAddElt (GnlFunctions (Gnl), (int)NewVar))
    return (GNL_MEMORY_FULL);
}

/* we delete all the nodes of 'Gnl'. */
GnlFreeNodesSegments (Gnl);

if (BListCreate (&NodesSegments))
    return (GNL_MEMORY_FULL);
SetGnlNodesSegments (Gnl, NodesSegments);
SetGnlFirstNode (Gnl, NULL);
SetGnlLastNode (Gnl, NULL);

if (BListCreate (&ListVariables))
    return (GNL_MEMORY_FULL);
for (i=0; i<BListSize (GnlInputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlInputs (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}
for (i=0; i<BListSize (GnlOutputs (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlOutputs (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}
for (i=0; i<BListSize (GnlLocals (Gnl)); i++)
{
    VarI = (GNL_VAR)BListElt (GnlLocals (Gnl), i);
    if (BListAddElt (ListVariables, (int)VarI))
        return (GNL_MEMORY_FULL);
}

for (i=0; i<BListSize (ListListCubes); i++)
{
    VarI = (GNL_VAR)BListElt (GnlFunctions (Gnl), i);
    FunctionI = GnlVarFunction (VarI);
    ListCubesI = (BLIST)BListElt (ListListCubes, i);
    if (GnlGetNodeFromListCubes (Gnl, ListVariables, ListCubesI,
                                &NewNode))
        return (GNL_MEMORY_FULL);
    SetGnlFunctionOnSet (FunctionI, NewNode);

    BListDelete (&ListCubesI, GnlCubeFree);
    ListCubesI = GnlFunctionDCSetCubes (FunctionI);
    BListDelete (&ListCubesI, GnlCubeFree);
    SetGnlFunctionOnSetCubes (FunctionI, NULL);
    SetGnlFunctionDCSetCubes (FunctionI, NULL);
}

```

lsfact.c

```
BListQuickDelete (&ListListCubes);
BListQuickDelete (&ListVariables);

    return (GNL_OK);
}

/*
 * LsFactorization
 */
GNL_STATUS LsFactorization (OriginalGnl, Gnl)
    GNL      OriginalGnl;
    GNL      Gnl;
{
    int                  i;
    LS_LOC_DIVISOR LocDivisorI;
    int                  DivisionDone;
    int                  ComplementDone;
    int                  ExtractDone;
    BLIST      ListKerDiv;
    BLIST      ListKerExtract;
    BLIST      ListCubes;
    BLIST      ListListCubes;
    BLIST      ListListKerDiv;

    if (BListCreate (&ListKerDiv))
        return (GNL_MEMORY_FULL);

    if (BListCreate (&ListKerExtract))
        return (GNL_MEMORY_FULL);

    if (BListCreate (&ListCubes))
        return (GNL_MEMORY_FULL);

    if (BListCreate (&ListListCubes))
        return (GNL_MEMORY_FULL);

    ListListKerDiv = NULL;
    DivisionDone = 0;
    ComplementDone = 0;
    ExtractDone = 0;

    /* We call NB_STEP_FACTORIZATION times the procedure below          */
    /* 'LsFactorizationStep' with a different TreshHold 'STEP_FACT[i]'   */
    for (i=0; i<NB_STEP_FACTORIZATION; i++)
    {
#endif TRACE_FACTORIZATION
        fprintf (stderr, " FACTORIZATION: Iteration [%d]\n", i);
#endif

        if (LsFactorizationStep (Gnl, &ListListCubes, &ListCubes,
                                &ListKerDiv, &ListListKerDiv,
                                &ListKerExtract, &DivisionDone,
                                &ComplementDone, &ExtractDone,
                                STEP_FACT[i]))
            return (GNL_MEMORY_FULL);
    }
}
```

```

    }

BListDelete (&ListKerDiv, LsGlobalDivisorFree);
BListDelete (&ListListKerDiv, BListQuickDelete);

for (i=0; i<BListSize (ListKerExtract); i++)
{
    LocDivisorI = (LS_LOC_DIVISOR)BListElt (ListKerExtract, i);
    BListQuickDelete (&LsLocDivisorKernel (LocDivisorI));
    GnlCubeFree (&LsLocDivisorCokernel (LocDivisorI));
    free ((char*)LocDivisorI);
}
BListQuickDelete (&ListKerExtract);

/* We rebuild a Gnl from the Cubes of 'Gnl'. */
if (LsBuildGnlFromGnlCubes (OriginalGnl, Gnl, ListListCubes))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

```

```

/*
 * LsFactorize
 */
GNL_STATUS LsFactorize (OriginalGnl, Gnl, OptPanel)
    GNL          OriginalGnl;
    GNL          Gnl;
    LS_OPT_PANEL OptPanel;
{
    G_OptPanel = OptPanel;
    G_NbVar = BListSize (GnlInputs (Gnl)) + BListSize (GnlOutputs (Gnl)) +
              BListSize (GnlLocals (Gnl));
    G_NbCells = (unsigned) NbOfCells (G_NbVar, BITS_PER_INT);

    if (LsFactorization (OriginalGnl, Gnl))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*
 * ----- EOF -----
 */

```

lsfact.h

```
/*
 *-----*/
/*          lsfact.h
 *-----*/
/*-----*/
/*-----*/
/*-----*/
/*-----*/
/*-----*/
/*-----*/
#ifndef LSFACT_H
#define LSFACT_H

#define LS_FACT_DIVISION      0
#define LS_FACT_EXTRACTION    1

/*-----*/
/* LS_GLOB_DIVISOR_REC */
/*-----*/
typedef struct LS_GLOB_DIVISOR_STRUCT {
    BLIST   Kernel;        /* List of GNL_CUBE
    BLIST   Cokernels;     /* List of LS_COKERNEL
}
LS_GLOB_DIVISOR_REC, *LS_GLOB_DIVISOR;

#define LsGlobDivisorKernel(d)           (((d)->Kernel))
#define LsGlobDivisorCokernels(d)        (((d)->Cokernels))

#define SetLsGlobDivisorKernel(d,k)      (((d)->Kernel = k))
#define SetLsGlobDivisorCokernels(d,c)   (((d)->Cokernels = c))

/*-----*/
/* LS_COKERNEL_REC */
/*-----*/
typedef struct LS_COKERNEL_STRUCT {
    int     Function;       /* */
    GNL_CUBE Cube;         /* */
}
LS_COKERNEL_REC, *LS_COKERNEL;

#define LsCokernelFunction(c)           (((c)->Function))
#define LsCokernelCube(c)              (((c)->Cube))

#define SetLsCokernelFunction(c,f)     (((c)->Function = f))
#define SetLsCokernelCube(c,co)        (((c)->Cube = co))

/*-----*/
/* LS_LOC_DIVISOR_REC */
/*-----*/
typedef struct LS_LOC_DIVISOR_STRUCT {
    BLIST   Kernel;        /* List of GNL_CUBE
    GNL_CUBE Cokernel;     /* The associated Cokernel
    int     Gain;
}
LS_LOC_DIVISOR_REC, *LS_LOC_DIVISOR;

#define LsLocDivisorKernel(d)           (((d)->Kernel))
#define LsLocDivisorCokernel(d)        (((d)->Cokernel))
```

lsfact.h

```
#define LsLocDivisorGain(d)          (((d)->Gain))  
  
#define SetLsLocDivisorKernel(d,k)    (((d)->Kernel = k))  
#define SetLsLocDivisorCokernel(d,c)  (((d)->Cokernel = c))  
#define SetLsLocDivisorGain(d,g)      (((d)->Gain = g))  
  
/*----- EOF -----*/  
  
#endif
```

lsfilter.c

```
/*
 *-----*/
/* File: lsfilter.c
 * Version: 1.1
 * Modifications: -
 * Documentation: -
 */
/* Class: none
 * Inheritance: -
 */
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnlcube.h"
#include "lsfact.h"
#include "gnloption.h"

/*
 * -----
 * EXTERN
 * -----
 * extern GNL_ENV G_GnlEnv;

/*
 * -----
 * LsDivGainFunction
 * -----
 * GNL_STATUS LsDivGainFunction (Gnl, ListListCubes, Kernel, ListCokernels,
 *                               NbCells, Gain)
 *   GNL           Gnl;
 *   BLIST         ListListCubes;
 *   BLIST         Kernel;
 *   BLIST         ListCokernels;
 *   int           NbCells;
 *   int           *Gain;
 {
   int           i;
   LS_COKERNEL  CokernelI;

   if (BListSize (Kernel) <= GnlEnvMinShareLogicSize ())
   {
     *Gain = 0;
     return (GNL_OK);
   }

   *Gain = BListSize (Kernel)*(BListSize(ListCokernels)-1);

   for (i=0; i<BListSize(ListCokernels); i++)
   {
     CokernelI = (LS_COKERNEL)BListElt (ListCokernels, i);
     *Gain += GnlCubeLitNumber (LsCokernelCube (CokernelI), NbCells) *
   }
}
```

lsfilter.c

```
        (BListSize (Kernel)-1);
    }

    return (GNL_OK);
}

/* -----
/* LsDivGainFunctionEqGates
/* -----
GNL_STATUS LsDivGainFunctionEqGates (Gnl, ListListCubes, Kernel,
ListCokernels,
NbCells, Gain)
{
    GNL          Gnl;
    BLIST        ListListCubes;
    BLIST        Kernel;
    BLIST        ListCokernels;
    int          NbCells;
    int          *Gain;
{
    int          i;
    GNL_CUBE    CubeI;
    int          NbEqGatesKernel;
    int          NbEqGatesCok;
    LS_COKERNEL CokernelI;

    NbEqGatesKernel = 0;
    NbEqGatesCok = 0;
    for (i=0; i<BListSize(Kernel); i++)
    {
        CubeI = (GNL_CUBE)BListElt (Kernel, i);
        NbEqGatesKernel += GnlCubeLitNumber (CubeI, NbCells);
    }
    for (i=0; i<BListSize(ListCokernels); i++)
    {
        CubeI = LsCokernelCube ((LS_COKERNEL)BListElt (ListCokernels, i));
        NbEqGatesCok += GnlCubeLitNumber (CubeI, NbCells);
    }

    *Gain = (BListSize(ListCokernels)-1)*NbEqGatesKernel +
           (BListSize (Kernel)-1)*NbEqGatesCok +
           BListSize (Kernel)-BListSize(ListCokernels)-2;
}

return (GNL_OK);
}

/* -----
/* LsDivChoiceFunction
/* -----
GNL_STATUS LsDivChoiceFunction (Gnl, ListListCubes, ListKernels,
IndexList, NbCells, Choice)
{
    GNL          Gnl;
    BLIST        ListListCubes;
    BLIST        ListKernels;
    BLIST        IndexList;
    int          NbCells;
    int          *Choice;
```

lsfilter.c

```
{  
    int i;  
    int IndexI;  
    LS_GLOB_DIVISOR BoxI;  
    BLIST ListEquations;  
    int MinCompute;  
  
    MinCompute = 1000000;  
  
    if (BListSize (IndexList) == 1)  
    {  
        *Choice = (int)BListElt (IndexList, 0);  
        return (GNL_OK);  
    }  
  
    for (i=0; i<BListSize (IndexList); i++)  
    {  
        IndexI = (int)BListElt (IndexList, i);  
        BoxI = (LS_GLOB_DIVISOR)BListElt (ListKernels, IndexI);  
        ListEquations = LsGlobDivisorCokernels (BoxI);  
  
        if (BListSize (ListEquations) < MinCompute)  
        {  
            *Choice = IndexI;  
            MinCompute = BListSize (ListEquations);  
        }  
    }  
  
    return (GNL_OK);  
}  
  
/* ----- */  
/* LsExtractGainFunction */  
/* ----- */  
GNL_STATUS LsExtractGainFunction (Gnl, ListListCubes, Kernel, CoKernel,  
                                  NbCells, Gain)  
{  
    GNL Gnl;  
    BLIST ListListCubes;  
    BLIST Kernel;  
    GNL_CUBE CoKernel;  
    int NbCells;  
    int *Gain;  
    {  
        int NbLitt;  
        int NbCube;  
  
        *Gain = 0;  
  
        NbCube = BListSize (Kernel);  
  
        if (BListSize (Kernel) == 1)  
            return (GNL_OK);  
  
        if (BListSize (Kernel) <= GnlEnvMinShareLogicSize ())  
            return (GNL_OK);  
    }
```

lsfilter.c

```
NbLitt = GnlCubeLitNumber (CoKernel, NbCells);

*Gain += NbLitt * NbCube - NbLitt - NbCube + 1;

return (GNL_OK);
}

/*
/* LsExtractGainFunctionEqGates
*/
/*
GNL_STATUS LsExtractGainFunctionEqGates (Gnl, ListListCubes, Kernel,
                                         CoKernel, NbCells, Gain)
{
    GNL          Gnl;
    BLIST        ListListCubes;
    BLIST        Kernel;
    GNL_CUBE    CoKernel;
    int          NbCells;
    int          *Gain;
{
    int          NbLitt;
    int          NbCube;

*Gain = 0;

if (BListSize (Kernel) == 1)
    return (GNL_OK);

NbCube = BListSize (Kernel);

NbLitt = GnlCubeLitNumber (CoKernel, NbCells);

*Gain = (NbLitt-1) * (NbCube-1);

return (GNL_OK);
}
/*
/* NbIntersect
*/
/*
int NbIntersect (Cube, ListCube)
{
    GNL_CUBE    Cube;
    BLIST        ListCube;

{
    int          i;
    int          Intersect;
    GNL_CUBE    CubeI;

Intersect =0;

if (ListCube == NULL)
    return (0);

for (i = 0; i < BListSize (ListCube); i++)
{
    CubeI = (GNL_CUBE)BListElt (ListCube, i);
    if (Cube == CubeI)
```

```

        Intersect++;
    }

    return (Intersect);
}

/*
 * LsExtractChoiceFunction
 */
GNL_STATUS LsExtractChoiceFunction (Gnl, ListListCubes, ListKer,
                                    IndexList, NbCells, BestIndex)
{
    GNL          Gnl;
    BLIST        ListListCubes;
    BLIST        ListKer;
    BLIST        IndexList;
    int          NbCells;
    int          *BestIndex;
{
    BLIST        Kernel;
    BLIST        Equation;
    int          i;
    int          k;
    int          Intersect;
    int          IntersectMin;
    int          IndexI;
    LS_LOC_DIVISOR BoxI;
    GNL_CUBE     CubeK;

    IntersectMin = 1000000;

    if (BListSize (IndexList) == 1)
    {
        *BestIndex = (int)BListElt (IndexList, 0);
        return (GNL_OK);
    }

    if (BListCreate (&Equation))
        return (GNL_MEMORY_FULL);

    for (i = 0; i < BListSize (ListKer); i++)
    {
        BoxI = (LS_LOC_DIVISOR)BListElt (ListKer, i);
        Kernel = LsLocDivisorKernel(BoxI);
        for (k = 0; k < BListSize (Kernel); k++)
        {
            CubeK = (GNL_CUBE)BListElt (Kernel, k);
            if (BListAddElt (Equation, (int)CubeK))
                return (GNL_MEMORY_FULL);
        }
    }

    for (i = 0; i < BListSize (IndexList); i++)
    {
        Intersect = 0;
        IndexI = (int)BListElt (IndexList, i);
        BoxI = (LS_LOC_DIVISOR)BListElt (ListKer, IndexI);

```

lsfilter.c

```

Kernel = LsLocDivisorKernel(BoxI);
for (k = 0; k < BListSize (Kernel); k++)
{
    CubeK = (GNL_CUBE)BListElt (Kernel, k);
    Intersect += NbIntersect (CubeK, Equation) - 1;
}
if (Intersect < IntersectMin)
{
    *BestIndex = IndexI;
    IntersectMin = Intersect;
}
}

BListQuickDelete (&Equation);

return (GNL_OK);
}

```

lsker.c

```
/*
 *-----*/
/* File: lsker.c
 * Version: 1.1
 * Modifications: -
 * Documentation: -
 */
/* Class: none
 * Inheritance:
 */
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "gnlmint.h"
#include "gnlcube.h"
#include "lsfact.h"
#include "gnlopt.h"

#include "blist.e"
/* -----
extern void GnlCubeFree ();
extern int GnlCubeIdentical ();
extern LS_OPT_PANEL G_OptPanel;

/* -----
#define MAX_CALL \
    (LsOptPanelDoOpt (G_OptPanel) == 2 ? 5000000 : 1000000)

/* -----
/* GLOBAL VARIABLES
 */
static int      G_NbKernelFound;
static GNL_CUBE   G_DivisorCube;
static int      G_MaxKernels; /* Maximum Nb divisors allowed */
static int      G_NbCells;
static int      G_NbVar;
static BLIST     G_ListKernels;
static BLIST     G_HashListKernels ;
static int      G_IndexFunction;
static int      G_MaxCall;
static int      G_MaxCallValue;

/* -----
/* LsCokernelFree
 */
void LsCokernelFree (Cokernel)
    LS_COKERNEL *Cokernel;
{
    if ((*Cokernel) == NULL)
        return;

    GnlCubeFree (&LsCokernelCube (*Cokernel));
}
```

lsker.c

```
free ((char*) (*Cokernel));
*Cokernel = NULL;

}

/*
/* LsGlobalDivisorFree
*/
void LsGlobalDivisorFree (GlobDivisor)
    LS_GLOB_DIVISOR      *GlobDivisor;
{

    if (LsGlobDivisorKernel (*GlobDivisor))
        BListDelete (&LsGlobDivisorKernel(*GlobDivisor), GnlCubeFree);

    if (LsGlobDivisorCokernels (*GlobDivisor))
        BListDelete (&LsGlobDivisorCokernels(*GlobDivisor),
                     LsCokernelFree);
    free ((char*) (*GlobDivisor));
    *GlobDivisor = NULL;
}

/*
/* GnlDivCommonDivisorCubeCube
*/
/* Gives to Cube 'Cube3' the value of common divisor between 'Cube1' and*/
/* 'Cube2'.
*/
void GnlDivCommonDivisorCubeCube (Cube1, Cube2, Cube3, NbCells)
    GNL_CUBE Cube1;
    GNL_CUBE Cube2;
    GNL_CUBE Cube3;
    int          NbCells;
{

    while (NbCells--)
    {
        GnlCubeHigh(Cube3) [NbCells] = GnlCubeHigh(Cube1) [NbCells] &
            GnlCubeHigh(Cube2) [NbCells];
        GnlCubeLow(Cube3) [NbCells] = GnlCubeLow(Cube1) [NbCells] |
            GnlCubeLow(Cube2) [NbCells];
    }
}

/*
/* GnlDivQuotientExistCubeCube
*/
/* Returns 1 if the division of 'M1' by 'M2' is not NULL, 0 otherwise */
int GnlDivQuotientExistCubeCube (M1, M2, NbCells)
    GNL_CUBE M1;
    GNL_CUBE M2;
    int          NbCells;
{
    int          k;
```

lsker.c

```
int M1var;
int M2var;

/* Are variables of 'M2' included in variables of 'M1' under normal */
/* form ? */
/* 1s of M2var are included in 1s of M1var iff M2var & ~M1var = 0 */
for (k = 0; k < NbCells; k++)
{
    M1var = GnlCubeHigh(M1) [k] & GnlCubeLow(M1) [k];
    M2var = GnlCubeHigh(M2) [k] & GnlCubeLow(M2) [k];
    if (M2var & ~M1var)
        return (0);
}

/* Are variables of 'M2' included in variables of 'M1' under comple- */
/* mented form ? */
/* 0s of M2var are included in 0s of M1var iff ~M2var & M1var = 0 */
for (k = 0; k < NbCells; k++)
{
    M1var = GnlCubeHigh(M1) [k] | GnlCubeLow(M1) [k];
    M2var = GnlCubeHigh(M2) [k] | GnlCubeLow(M2) [k];
    if (~M2var & M1var)
        return (0);
}

return (1);
}

/*
 * GnlDivQuickQuotientCubeCube
 */
/*
 * performs quick division of 'M1' divided by 'M2' and stored in 'M3'
 */
void GnlDivQuickQuotientCubeCube (M1, M2, M3, NbCells)
    GNL_CUBE M1;
    GNL_CUBE M2;
    GNL_CUBE M3;
    int NbCells;
{
    int M2var;

    while (NbCells--)
    {
        M2var = ~(GnlCubeHigh(M2) [NbCells] ^ GnlCubeLow(M2) [NbCells]);
        GnlCubeHigh(M3) [NbCells] = GnlCubeHigh(M1) [NbCells] & ~M2var;
        GnlCubeLow(M3) [NbCells] = GnlCubeLow(M1) [NbCells] | M2var;
    }
}

/*
 * GnlDivQuotientListCubes
 */
/*
 * Performs the division of 'F' by 'Divisor' and stores the result in
 * 'Q'. 'Q' must have been pre-allocated.
 */

```

```

GNL_STATUS GnlDivQuotientListCubes (F, Divisor, Q, NbCells)
    BLIST      F;
    GNL_CUBE  Divisor;
    BLIST      Q;
    int        NbCells;
{
    int          i;
    GNL_CUBE  CubeI;
    GNL_CUBE  Cube;

    for (i = 0; i < BListSize (F); i++)
    {
        CubeI = (GNL_CUBE)BListElt (F, i);
        if (GnlDivQuotientExistCubeCube (CubeI, Divisor, NbCells))
        {
            if (GnlCubeFullCreate (NbCells, &Cube))
                return (GNL_MEMORY_FULL);

            GnlDivQuickQuotientCubeCube (CubeI, Divisor, Cube, NbCells);

            if (BListAddElt (Q, (int)Cube))
                return (GNL_MEMORY_FULL);
        }
    }

    return (GNL_OK);
}

/*
/* LsGetKernelHashKey
*/
int LsGetKernelHashKey (Kernel, NbCells)
    BLIST      Kernel;
    int        NbCells;
{
    int          i;
    int          NbLit;
    GNL_CUBE  CubeI;

    NbLit = 0;

    for (i=0; i<BListSize (Kernel); i++)
    {
        CubeI = (GNL_CUBE)BListElt (Kernel, i);
        NbLit += GnlCubeLitNumber (CubeI, NbCells);
    }

    return (20 * NbLit + BListSize (Kernel));
/*
    return (BListSize (Kernel));
*/
}
/*
/* LsEquivalentKernel
*/

```

lsker.c

```
/* ----- */
int LsEquivalentKernel (Kernel1, Kernel2, NbCells)
    BLIST    Kernel1;
    BLIST    Kernel2;
    int      NbCells;
{

    return (BListContextualIdentical (Kernel1, Kernel2, GnlCubeIdentical,
                                      NbCells));
}

/* ----- */
/* LsAddOrGetKernelInHashTable */
/* ----- */
GNL_STATUS LsAddOrGetKernelInHashTable (Kernel, Cokernel, IndexFunction,
                                         Added)
    BLIST    Kernel;
    GNL_CUBE Cokernel;
    int      IndexFunction;
    int      *Added;
{
    int          i;
    int          Key;
    BLIST        KernelsBucket;
    LS_GLOB_DIVISOR   DivisorI;
    LS_GLOB_DIVISOR   NewDivisor;
    BLIST        NewList;
    LS_COKERNEL    NewCokernel;

    /* Build a LS_COKERNEL of (IndexFunction, Cokernel). */
    if ((NewCokernel = (LS_COKERNEL)
         calloc (1, sizeof(LS_COKERNEL_REC))) ==NULL)
        return (GNL_MEMORY_FULL);
    SetLsCokernelFunction (NewCokernel, IndexFunction);
    SetLsCokernelCube (NewCokernel, Cokernel);

    /* Compute the Key of the kernel 'Kernel'. */
    Key = LsGetKernelHashKey (Kernel, G_NbCells) %
          BListSize(G_HashListKernels);

    KernelsBucket = (BLIST)BListElt (G_HashListKernels, Key);

    for (i=0; i<BListSize (KernelsBucket); i++)
    {
        DivisorI = (LS_GLOB_DIVISOR)BListElt (KernelsBucket, i);
        if (LsEquivalentKernel (Kernel, LsGlobDivisorKernel(DivisorI),
                               G_NbCells))
        {
            if (BListAddElt (LsGlobDivisorCokernels(DivisorI),
                            (int)NewCokernel))
                return (GNL_MEMORY_FULL);
            BListDelete (&Kernel, GnlCubeFree);
        }
        *Added = 0;
```

```

        return (GNL_OK);
    }

/* Otherwise we need to create a new Divisor */ 
if ((NewDivisor = (LS_GLOB_DIVISOR)
    calloc (1, sizeof(LS_GLOB_DIVISOR_REC))) == NULL)
return (GNL_MEMORY_FULL);

SetLsGlobDivisorKernel (NewDivisor, Kernel);
if (BListCreateWithSize (1, &NewList))
    return (GNL_MEMORY_FULL);
if (BListAddElt (NewList, (int)NewCokernel))
    return (GNL_MEMORY_FULL);
SetLsGlobDivisorCokernels (NewDivisor, NewList);

/* adding the new divisor in the Hash list of Kernels */
if (BListElt (G_HashListKernels, Key) == (int)NULL)
{
    if (BListCreateWithSize (1, &NewList))
        return (GNL_MEMORY_FULL);
    BListElt (G_HashListKernels, Key) = (int)NewList;
}
NewList = (BLIST)BListElt (G_HashListKernels, Key);
if (BListAddElt (NewList, (int)NewDivisor))
    return (GNL_MEMORY_FULL);

/* Adding the new divisor in the List of Kernels */
if (BListAddElt (G_ListKernels, (int)NewDivisor))
    return (GNL_MEMORY_FULL);

*Added = 1;

return (GNL_OK);
}

/*
 * LsExistKernel
 */
/*
 * This function returns 1 if there exists a Kernel in the list of cubes*/
/* A kernel must be composed at least of two Cubes. 'NewCube' will */
/* correspond to the associated Cokernel. */
/*
int LsExistKernel (ListCubes, Index, NewCube)
    BLIST    ListCubes;
    int      Index;
    GNL_CUBE NewCube;
{
    int          i;
    int          NbCube;
    GNL_CUBE CubeI;
    int          t;
    int          d;
    unsigned mask;

```

lsker.c

```
NbCube = 0;
t = (Index >> 1) % BITS_PER_INT;
d = (Index >> 1) / BITS_PER_INT;
mask = 1 << t;

for (i = 0; i < BListSize (ListCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (ListCubes, i);
    if (GnlCubeLiteralExistQuick (CubeI, Index, t, d, mask))
    {
        if (!NbCube)
            GnlCubeInitWithValue (NewCube, "?", G_NbCells);

        GnlDivCommonDivisorCubeCube (NewCube, CubeI, NewCube,
                                      G_NbCells);
        NbCube++;
    }
}

return (NbCube >= 2);
}

/*
 * LsLocateFirstBit1
 */
int LsLocateFirstBit1 (Val)
{
    int      Val;
    int      i;

    i = 1;
    while (i <= BITS_PER_INT)
    {
        if ((1 << (i - 1)) & (Val))
            return (i);
        i++;
    }

    return (0);
}

/*
 * LsFirstCubeLiteral
 */
/* This function returns 1 if the first literal of 'Cube' is the 'Index'*/
/* one. Returns 0 otherwise. */
int LsFirstCubeLiteral (Index, Cube)
{
    int      Index;
    GNL_CUBE Cube;
    int      i;
    int      Test;
    int      Limit;

    Limit = (Index / 2) / BITS_PER_INT;
```

```

/* Bits at 1 of 'Test' tells the corresponding variable is in 'Cube' */
for (i = 0; i < Limit; i++)
{
    Test = ~ (GnlCubeHigh(Cube) [i] ^ GnlCubeLow(Cube) [i]);

    /* If yes then the Cokernel is already known. */
    if (Test)
        return (0);
}

/* i = Limit;
Test = ~ (GnlCubeHigh(Cube) [i] ^ GnlCubeLow(Cube) [i]); */

if (LsLocateFirstBit1 (Test) < ((Index / 2) % BITS_PER_INT) + 1)
    return (0);

return (1);
}

/*
 * LsGetGlobalKernels
 */
/*
 * Recursive procedure which computes all the Kernel and associated
 * Cokernel from 'LastKernel' at index 'StartIndex'.
 */
GNL_STATUS LsGetGlobalKernels (LastKernel, KernelSupport, StartIndex,
                               LastCokernel)
{
    BLIST     LastKernel;
    unsigned *KernelSupport;
    int       StartIndex;
    GNL_CUBE LastCokernel;

    int       Index;
    BLIST     NewKernel;
    GNL_CUBE NewCokernel;
    int       Added;
    int       Limit;

    Limit = 2 * G_NbVar;

    for (Index = StartIndex; Index < Limit; Index++)
    {
        /* Actually this var. is not in the Support of 'LastKernel' */
        if (KernelSupport && !MintBitValue (KernelSupport, (Index/2)+1))
        {
            Index++;
            continue;
        }

        G_MaxCall += BListSize (LastKernel);
        if (G_MaxCall > G_MaxCallValue)
        {
            BListDelete (&LastKernel, GnlCubeFree);
            GnlCubeFree (&LastCokernel);
            return (GNL_OK);
        }
    }
}

```

```

    }

/* We check if there exists a sub-kernel in 'LastKernel' which */
/* starts from literal at index 'Index'. If yes we verify that */
/* the corresponding Cokernel 'G_DivisorCube' has as first */
/* literal the one of index 'Index'. If not, this means that */
/* kernel has been already processed. */
if (LsExistKernel (LastKernel, Index, G_DivisorCube) &&
    LsFirstCubeLiteral (Index, G_DivisorCube))
{
    if (BListCreateWithSize (2, &NewKernel))
        return (GNL_MEMORY_FULL);

    GnlDivQuotientListCubes (LastKernel, G_DivisorCube,
                             NewKernel, G_NbCells);

    if (GnlCubeCopy (G_NbCells, LastCokernel, &NewCokernel))
        return (GNL_MEMORY_FULL);

    GnlCubeMultQuickCubeCube (NewCokernel, G_DivisorCube,
                             NewCokernel, G_NbCells);

    if (LsGetGlobalKernels (NewKernel, NULL, Index + 2 - Index % 2,
                           NewCokernel))
        return (GNL_MEMORY_FULL);

    if (G_NbKernelFound > G_MaxKernels)
    {
        BListDelete (&LastKernel, GnlCubeFree);
        GnlCubeFree (&LastCokernel);
        return (GNL_OK);
    }
}

/* We try to add this new kernel in the global hash List */
/* 'HashListKernels' and in 'ListKernels' */
if (LsAddOrGetKernelInHashTable (LastKernel, LastCokernel,
                                 G_IndexFunction, &Added))
    return (GNL_MEMORY_FULL);

/* 'Added' is 1 if a new Kernel has been added and 0 otherwise */
G_NbKernelFound += Added;

return (GNL_OK);
}

/*
 * -----
 * LsGetListCubesSupport
 * -----
 * This procedure creates a *unsigned representing the support of the
 * variables of the list of cubes 'ListCubes'. A bit to 1 means that the
 * variable is present.
 * -----
 * GNL_STATUS LsGetListCubesSupport (ListCubes, NbCells, CubeSupport)
 */

```

lsker.c

```
BLIST      ListCubes;
int         NbCells;
unsigned **CubeSupport;
{
    GNL_CUBE     CubeI;
    int          i;
    int          Cells;

    if (MintCreateInit0 (NbCells, CubeSupport))
        return (GNL_MEMORY_FULL);

    for (i=0; i<BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        Cells = NbCells;
        while (Cells--)
        {
            (*CubeSupport) [Cells] |= ~(GnlCubeHigh(CubeI) [Cells] ^
                                         GnlCubeLow(CubeI) [Cells]);
        }
    }

    return (GNL_OK);
}

/* -----
/* LsExtractGlobalKernels
/* -----
GNL_STATUS LsExtractGlobalKernels (ListCubes, IndexFunction, ListKernels,
                                    HashListKernels, MaxKernels,
                                    NbVar, NbCells)
{
    BLIST      ListCubes;
    int         IndexFunction;
    BLIST      ListKernels;
    BLIST      HashListKernels;
    int         MaxKernels;
    int         NbVar;
    int         NbCells;
{
    int          i;
    GNL_CUBE    CubeI;
    BLIST       Kernel;
    GNL_CUBE    Cokernel;
    unsigned **KernelSupport;

    /* Setting global variables in order to not propagate them thru
    /* recursive calls.                                                 */
    G_NbKernelFound = 0;
    G_MaxKernels = MaxKernels;
    G_NbCells = NbCells;
    G_NbVar = NbVar;
    G_MaxCall = 0;
    G_MaxCallValue = MAX_CALL;
```

```

G_ListKernels = ListKernels;
G_HashListKernels = HashListKernels;
G_IndexFunction = IndexFunction;
if (GnlCubeFullCreate (G_NbCells, &G_DivisorCube))
    return (GNL_MEMORY_FULL);
GnlCubeInitWithValue (G_DivisorCube, "?", G_NbCells);

for (i=0; i<BListSize (ListCubes); i++)
{
    CubeI = (GNL_CUBE)BListElt (ListCubes, i);
    GnlDivCommonDivisorCubeCube (G_DivisorCube, CubeI, G_DivisorCube,
                                  NbCells);
}

/* Creating the first Kernel. */
if (BListCreateWithSize (1, &Kernel))
    return (GNL_MEMORY_FULL);
if (GnlDivQuotientListCubes (ListCubes, G_DivisorCube, Kernel, NbCells))
    return (GNL_MEMORY_FULL);

/* Duplicating the first Cokernel. */
if (GnlCubeCopy (NbCells, G_DivisorCube, &Cokernel))
    return (GNL_MEMORY_FULL);

if (LsGetListCubesSupport (Kernel, NbCells, &KernelSupport))
    return (GNL_MEMORY_FULL);

/* Running recursively the computation */
if (LsGetGlobalKernels (Kernel, KernelSupport, 0, Cokernel))
    return (GNL_MEMORY_FULL);

GnlCubeFree (&G_DivisorCube);
free (KernelSupport);

return (GNL_OK);
}

/*
 * GnlQuickDivision
 */
GNL_STATUS GnlQuickDivision (ListCubes, Cube, ListCubesIncluded, NbCells)
    BLIST ListCubes;
    GNL_CUBE Cube;
    BLIST ListCubesIncluded;
    int NbCells;
{
    int i;
    GNL_CUBE CubeI;

    for (i=0; i<BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        if (GnlCubeIncluded (NbCells, CubeI, Cube))
            if (BListAddElt (ListCubesIncluded, (int)CubeI))
                return (GNL_MEMORY_FULL);
    }
}

```

```

    }

    return (GNL_OK);
}

/* -----
/* LsAddLocalKernel
/* -----
GNL_STATUS LsAddLocalKernel (Kernel, Cokernel)
BLIST Kernel;
GNL_CUBE Cokernel;
{
    LS_LOC_DIVISOR NewLocDivisor;

    if ((NewLocDivisor = (LS_LOC_DIVISOR)
        calloc (1, sizeof (LS_LOC_DIVISOR_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetLsLocDivisorKernel (NewLocDivisor, Kernel);
    SetLsLocDivisorCokernel (NewLocDivisor, Cokernel);

    if (BListAddElt (G_ListKernels, (int)NewLocDivisor))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/* -----
/* LsExistLocalKernel
/* -----
GNL_STATUS LsExistLocalKernel (ListCubes, Index, Cube, ListCubesIncluded,
                               Exist)
BLIST ListCubes;
int Index;
GNL_CUBE Cube;
BLIST *ListCubesIncluded;
int *Exist;
{
    int i;
    int NbCube;
    GNL_CUBE CubeI;
    int t;
    int d;
    unsigned mask;

    NbCube = 0;
    *Exist = 0;
    t = (Index >> 1) % BITS_PER_INT;
    d = (Index >> 1) / BITS_PER_INT;
    mask = 1 << t;

    for (i = 0; i < BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE) BListElt (ListCubes, i);

```

```

    if (GnlCubeLiteralExistQuick (CubeI, Index, t, d, mask))
    {
        if (NbCube == 0)
        {
            GnlCubeInitWithValue (Cube, "?", G_NbCells);
            if (BListCreateWithSize (1, ListCubesIncluded))
                return (GNL_MEMORY_FULL);
        }

        GnlDivCommonDivisorCubeCube (Cube, CubeI, Cube, G_NbCells);

        NbCube++;
        if (BListAddElt (*ListCubesIncluded, CubeI))
            return (GNL_MEMORY_FULL);
    }
}

if ((NbCube == 1) || (NbCube == BListSize (ListCubes)))
{
    BListQuickDelete (ListCubesIncluded);
    *Exist = 0;
    return (GNL_OK);
}

*Exist = (NbCube != 0);

if ((NbCube == 1) && !(*Exist))
    BListQuickDelete (ListCubesIncluded);

return (GNL_OK);
}

/*
 * LsNewLocalKernel
 */
int LsNewLocalKernel (Index, Cube, LastCokernel)
    int             Index;
    GNL_CUBE       Cube;
    GNL_CUBE       LastCokernel;
{
    int          i;
    int          Limit;

    Limit = Index/2;
    for (i=0; i<Limit; i++)
    {
        if ((MintBitValue (GnlCubeLow(Cube), i+1) ==
            MintBitValue (GnlCubeHigh(Cube), i+1)) &&
            (MintBitValue (GnlCubeLow(LastCokernel), i+1) !=
            MintBitValue (GnlCubeHigh(LastCokernel), i+1)))
            return (0);
    }

    return (MintBitValue (GnlCubeLow(LastCokernel), i+1) !=
            MintBitValue (GnlCubeHigh(LastCokernel), i+1));
}

```

```

/*
 * ----- */
/* LsGetLocalAllKernels */
/* ----- */
GNL_STATUS LsGetLocalAllKernels (LastKernel, KernelSupport, StartIndex,
                                LastCokernel)
{
    BLIST     LastKernel;
    unsigned   *KernelSupport;
    int        StartIndex;
    GNL_CUBE  LastCokernel;
{
    int        Index;
    BLIST     NewKernel;
    GNL_CUBE  NewCokernel;
    int        Exist;
    int        Limit;

    if (GnlCubeLitNumber (LastCokernel, G_NbCells) > 1)
    {
        if (LsAddLocalKernel (LastKernel, LastCokernel))
            return (GNL_MEMORY_FULL);
        G_NbKernelFound++;
        if (G_NbKernelFound > G_MaxKernels)
        {
            return (GNL_OK);
        }
    }

    Limit = 2 * G_NbVar;
    for (Index = StartIndex; Index < Limit; Index++)
    {
        /* Actually this var. is not in the Support of 'LastKernel'      */
        if (KernelSupport && !MintBitValue (KernelSupport, (Index/2)+1))
        {
            Index++;
            continue;
        }

        G_MaxCall += BListSize (LastKernel);
        if (G_MaxCall > G_MaxCallValue)
        {
            return (GNL_OK);
        }

        if (LsExistLocalKernel (LastKernel, Index, G_DivisorCube,
                               &NewKernel, &Exist))
            return (GNL_MEMORY_FULL);

        if (Exist && LsNewLocalKernel (Index, G_DivisorCube, LastCokernel))
        {
            if (GnlCubeCopy (G_NbCells, G_DivisorCube, &NewCokernel))
                return (GNL_MEMORY_FULL);

            if (LsGetLocalAllKernels (NewKernel, NULL, Index + 2 - Index % 2,
                                     NewCokernel))
                return (GNL_MEMORY_FULL);
        }
    }
}

```

lsker.c

```
        }

    }

    if (GnlCubeLitNumber (&LastCokernel, G_NbCells) <= 1)
    {
        GnlCubeFree (&LastCokernel);
        BListQuickDelete (&LastKernel);
    }

    return (GNL_OK);
}

/* -----
/* LsExtractLocalKernels
/* ----- */
GNL_STATUS LsExtractLocalKernels (ListCubes, ListKernels, MaxKernels,
                                  NbVar, NbCells, GainFunction, Gnl,
                                  LOfLPT)
{
    BLIST      ListCubes;
    BLIST      ListKernels;
    int        MaxKernels;
    int        NbVar;
    int        NbCells;
    int        (*GainFunction) ();
    GNL        Gnl;
    BLIST      LOfLPT;

    int        i;
    GNL_CUBE   CubeI;
    BLIST      Kernel;
    GNL_CUBE   Cokernel;
    int        Gain;
    LS_LOC_DIVISOR LocalDivisor;
    unsigned   *KernelSupport;

    /* Setting global variables in order to not propagate them thru
     * recursive calls. */
    if (BListCreateWithSize (1, &G_ListKernels))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (NbVar, &G_HashListKernels))
        return (GNL_MEMORY_FULL);

    G_NbKernelFound = 0;
    G_MaxKernels = MaxKernels;
    G_NbCells = NbCells;
    G_NbVar = NbVar;
    G_MaxCall = 0;
    G_MaxCallValue = MAX_CALL;
    if (GnlCubeFullCreate (G_NbCells, &G_DivisorCube))
        return (GNL_MEMORY_FULL);
    GnlCubeInitWithValue (G_DivisorCube, "?", NbCells);

    for (i=0; i<BListSize (ListCubes); i++)
    {
        CubeI = (GNL_CUBE)BListElt (ListCubes, i);
        GnlDivCommonDivisorCubeCube (G_DivisorCube, CubeI, G_DivisorCube,
```

lsker.c

```
NbCells);  
}  
  
/* Creating the first Kernel. */  
if (BListCreateWithSize (1, &Kernel))  
    return (GNL_MEMORY_FULL);  
if (GnlQuickDivision (ListCubes, G_DivisorCube, Kernel, NbCells))  
    return (GNL_MEMORY_FULL);  
  
/* Duplicating the first Cokernel. */  
if (GnlCubeCopy (NbCells, G_DivisorCube, &Cokernel))  
    return (GNL_MEMORY_FULL);  
  
if (LsGetListCubesSupport (Kernel, NbCells, &KernelSupport))  
    return (GNL_MEMORY_FULL);  
  
/* Running recursively the computation */  
if (LsGetLocalAllKernels (Kernel, KernelSupport, 0, Cokernel))  
    return (GNL_MEMORY_FULL);  
  
GnlCubeFree (&G_DivisorCube);  
  
for (i=0; i<BListSize (G_ListKernels); i++)  
{  
    LocalDivisor = (LS_LOC_DIVISOR)BListElt (G_ListKernels, i);  
    Kernel = LsLocDivisorKernel (LocalDivisor);  
    Cokernel = LsLocDivisorCokernel (LocalDivisor);  
  
    if ((*GainFunction) (Gnl, LOFLPT, Kernel, Cokernel, NbCells,  
                        &Gain))  
        return (GNL_MEMORY_FULL);  
  
    SetLsLocDivisorGain (LocalDivisor, Gain);  
}  
  
/* We add the newlocal divisors to the previous stored ones */  
if (BListAppend (ListKernels, &G_ListKernels))  
    return (GNL_MEMORY_FULL);  
  
BListDelete (&G_HashListKernels, BListQuickDelete);  
  
free (KernelSupport);  
  
return (GNL_OK);  
}  
  
/* ----- */
```

skiplist.c

```
/*-----*/
/*
 *      File:          skiplist.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 *
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#include <stdio.h>
#include <stdlib.h>

#include "skiplist.h"

/*
 * -----
 *      salloc
 * -----
 * This function allocates memory for one segment of the skiplist nodes
 * and returns a pointer to the allocated memory or NULL in case
 * of failure. The memory is set to zero.
 */
static SKIPLIST_NODE salloc (void)
{
    return ((SKIPLIST_NODE) calloc (SKIPLIST_NODE_SEGMENT_SIZE,
                                    sizeof (SKIPLIST_NODE_REC)));
}

/*
 * -----
 *      SkipListFreeSegment
 * -----
 * static void SkipListFreeSegment (Segment)
 *     SKIPLIST_NODE *Segment;
 {
    free (*Segment);
    *Segment = NULL;
 }

/*
 * -----
 *      SkipListCreateNode
 * -----
 * Each created SKIPLIST_NODE object is relative to a 'SkipList' and
 * belongs to a segment of SKIPLIST_NODE.
 * You cannot free directly a specific SKIPLIST_NODE created in
 * this way, because then you free the whole segment.
 */
static SKIPLIST_NODE SkipListCreateNode (SkipList)
    SKIPLIST        SkipList;
{
    SKIPLIST_NODE   NewNode;
    int            Size;

    if (Size = BListSize (SkipListUsedNodes (SkipList)))
    {
```

skiplist.c

```
NewNode = (SKIPLIST_NODE)BListElt (SkipListUsedNodes (SkipList), Size-1);
    BListDelShift (SkipListUsedNodes (SkipList), Size);
    return (NewNode);
}

if (SkipListFirstNode (SkipList) >= SkipListLastNode (SkipList))
{
    /* We create a new segment of Nodes. */
    if ((NewNode = salloc ()) == NULL)
        return (NULL);
    SetSkipListFirstNode (SkipList, NewNode);
    SetSkipListLastNode (SkipList, NewNode+SKIPLIST_NODE_SEGMENT_SIZE);
    if (BListAddElt (SkipListNodesSegments (SkipList), (int)NewNode))
        return (NULL);
}
NewNode = SkipListFirstNode (SkipList);
SetSkipListFirstNode (SkipList, NewNode+1);

return (NewNode);
}

/* -----
/* SkipListFreeNode
/* ----- */
static int SkipListFreeNode (SkipList, Node)
    SKIPLIST      SkipList;
    SKIPLIST_NODE  Node;
{
    SetSkipListNodeKeyPtr (Node, NULL);
    SetSkipListNodeDataPtr (Node, NULL);
    SetSkipListNodeNext (Node, NULL);
    SetSkipListNodeDown (Node, NULL);
    return (BListAddElt (SkipListUsedNodes (SkipList), (int)Node));
}

/* -----
/* SkipListInsertRec
/* ----- */
static int SkipListInsertRec (SkipList, Node, KeyPtr, DataPtr, FirstNode)
    SKIPLIST      SkipList;
    SKIPLIST_NODE  Node;
    KEY_PTR       KeyPtr;
    DATA_PTR      DataPtr;
    SKIPLIST_NODE *FirstNode;
{
    SKIPLIST_NODE CurrNode = Node;
    SKIPLIST_NODE NewNode;
    SKIPLIST_NODE NewDown;
    SKIPLIST_NODE ThdSon;
    SKIPLIST_NODE FthSon;
    IntFctPtr     KeyCompare = SkipListKeyCompare (SkipList);
    int           Status;

    while (SkipListNodeNext (CurrNode) &&
          KeyCompare (KeyPtr,
                      skipListNodeKeyPtr (skipListNodeNext (CurrNode))) >= 0)
```

skiplist.c

```
CurrNode = SkipListNodeNext (CurrNode);

if (SkipListNodeIsLeaf (CurrNode))
{
    if (!KeyCompare (KeyPtr, SkipListNodeKeyPtr (CurrNode)))
        return (-2); /* The key pointed by KeyPtr is already in the tree */

    if (!(NewNode = SkipListCreateNode (SkipList)))
        return (-1);

    SkipListSizeIncr (SkipList);

    SetSkipListNodeKeyPtr (NewNode, KeyPtr);
    SetSkipListNodeDataPtr (NewNode, DataPtr);
    if (KeyCompare (KeyPtr, SkipListNodeKeyPtr (CurrNode)) > 0)
    {
        SetSkipListNodeNext (NewNode, SkipListNodeNext (CurrNode));
        SetSkipListNodeNext (CurrNode, NewNode);
        *FirstNode = Node;
    }
    else
    {
        SetSkipListNodeNext (NewNode, CurrNode);
        *FirstNode = NewNode;
    }
    return (1);
}

*FirstNode = Node;

Status = SkipListInsertRec (SkipList, SkipListNodeDown (CurrNode),
                            KeyPtr, DataPtr, &NewDown);

if (Status < 0)
    return (Status);

if (SkipListNodeDown (CurrNode) != NewDown)
    SetSkipListNodeDown (CurrNode, NewDown);
SetSkipListNodeKeyPtr (CurrNode,
                      SkipListNodeKeyPtr (SkipListNodeDown (CurrNode)));

if (!Status)
    return (0);

ThdSon = SkipListNodeNext (SkipListNodeNext (SkipListNodeDown (CurrNode)));

if (ThdSon && (FthSon = SkipListNodeNext (ThdSon)) &&
    (!SkipListNodeNext (CurrNode) ||
     (KeyCompare (SkipListNodeKeyPtr (FthSon),
                  SkipListNodeKeyPtr (SkipListNodeNext (CurrNode))) < 0)))
{
    if (!(NewNode = SkipListCreateNode (SkipList)))
        return (-1);
    SetSkipListNodeKeyPtr (NewNode, SkipListNodeKeyPtr (ThdSon));
    SetSkipListNodeNext (NewNode, SkipListNodeNext (CurrNode));
    SetSkipListNodeNext (CurrNode, NewNode);
    SetSkipListNodeDown (NewNode, ThdSon);
```

skiplist.c

```
    return (1);
}

return (0);
}

/*
 * SkipListRemoveRec
 */
/* This function removes the node with the key, pointed by the given
 * KeyPtr. If there was something wrong, -1 is returned. If the key,
 * pointed by the given KeyPtr, was in the skip list, in the
 * parameters NodesKeyPtr and NodesDataPtr the corresponding node's
 * KeyPtr and DataPtr are returned and: if no node was removed from
 * this level, 0 is returned, else 2 is returned. If the key, pointed
 * by the given KeyPtr, wasn't in the skip list, 1 is returned.
 */
static int SkipListRemoveRec (SkipList, Node, PrevNode, KeyPtr,
                           NodesKeyPtr, NodesDataPtr, FirstNode)
SKIPLIST           SkipList;
SKIPLIST_NODE      Node;
SKIPLIST_NODE      PrevNode;
KEY_PTR            KeyPtr;
KEY_PTR            *NodesKeyPtr;
DATA_PTR           *NodesDataPtr;
SKIPLIST_NODE      *FirstNode;
{
    SKIPLIST_NODE     CurrNode = Node;
    SKIPLIST_NODE     NewPrevNode = NULL;
    SKIPLIST_NODE     PrevSon = NULL;
    SKIPLIST_NODE     NewDown;
    SKIPLIST_NODE     SdSon;
    SKIPLIST_NODE     ThdSon;
    SKIPLIST_NODE     Brother;
    IntFctPtr         KeyCompare = SkipListKeyCompare (SkipList);
    int               Status;

    while (SkipListNodeNext (CurrNode) &&
          KeyCompare (KeyPtr,
                      SkipListNodeKeyPtr (SkipListNodeNext (CurrNode))) >= 0)
    {
        NewPrevNode = CurrNode;
        CurrNode = SkipListNodeNext (CurrNode);
    }

    if (SkipListNodeIsLeaf (CurrNode))
    {
        if (KeyCompare (KeyPtr, SkipListNodeKeyPtr (CurrNode)))
            return (1); /* The key pointed by the KeyPtr isn't in the list */

        *NodesKeyPtr = SkipListNodeKeyPtr (CurrNode);
        *NodesDataPtr = SkipListNodeDataPtr (CurrNode);

        if (NewPrevNode)
        {
            SetSkipListNodeNext (NewPrevNode, SkipListNodeNext (CurrNode));
        *FirstNode = Node;
    }
}
```

skiplist.c

```
    }
else
{
    *FirstNode = SkipListNodeNext (Node);
    if (PrevNode)
        SetSkipListNodeNext (PrevNode, SkipListNodeNext (Node));
}

if (SkipListFreeNode (SkipList, CurrNode))
    return (-1);

SkipListSizeDecr (SkipList);

return (2);
}

*FirstNode = Node;

if (NewPrevNode)
    PrevNode = NewPrevNode;

if (PrevNode)
{
    PrevSon =
        SkipListNodeNext (SkipListNodeNext (SkipListNodeDown (PrevNode)));
    if (KeyCompare (SkipListNodeKeyPtr (PrevSon),
                    SkipListNodeKeyPtr (CurrNode)) >= 0)
        PrevSon = SkipListNodeNext (SkipListNodeDown (PrevNode));
}

Status = SkipListRemoveRec (SkipList, SkipListNodeDown (CurrNode), PrevSon,
                            KeyPtr, NodesKeyPtr, NodesDataPtr, &NewDown);

if (Status != 1)
{
    if (SkipListNodeDown (CurrNode) != NewDown)
        SetSkipListNodeDown (CurrNode, NewDown);
    SetSkipListNodeKeyPtr (CurrNode,
                           SkipListNodeKeyPtr (SkipListNodeDown (CurrNode)));
}

if (Status < 2)
    return (Status);

if ((SdSon = SkipListNodeNext (SkipListNodeDown (CurrNode))) &&
    (KeyCompare (SkipListNodeKeyPtr (SdSon),
                 SkipListNodeKeyPtr (SkipListNodeNext (CurrNode))) < 0))
    return (0);

if (Brother = NewPrevNode)
{
    SdSon = SkipListNodeNext (SkipListNodeDown (Brother));
    ThdSon = SkipListNodeNext (SdSon);

    if (!KeyCompare (SkipListNodeKeyPtr (ThdSon),
                    SkipListNodeKeyPtr (CurrNode)))
    {
```

skiplist.c

```
/* Brother has only two sons, so we union them under Brother */  
SetSkipListNodeNext (Brother, SkipListNodeNext (CurrNode));  
if (SkipListFreeNode (SkipList, CurrNode))  
    return (-1);  
return (2);  
}  
  
/* Brother has three sons, so we borrow one from it */  
SetSkipListNodeDown (CurrNode, ThdSon);  
SetSkipListNodeKeyPtr (CurrNode, SkipListNodeKeyPtr (ThdSon));  
return (0);  
}  
  
Brother = SkipListNodeNext (CurrNode);  
SdSon = SkipListNodeNext (SkipListNodeDown (Brother));  
ThdSon = SkipListNodeNext (SdSon);  
  
if (ThdSon &&  
    (!SkipListNodeNext (Brother) ||  
     (KeyCompare (SkipListNodeKeyPtr (ThdSon),  
                  SkipListNodeKeyPtr (SkipListNodeNext (Brother))) < 0)))  
{  
    /* Brother has three sons, so we borrow one from it */  
    SetSkipListNodeDown (Brother, SdSon);  
    SetSkipListNodeKeyPtr (Brother, SkipListNodeKeyPtr (SdSon));  
    return (0);  
}  
  
/* Brother has only two sons, so we union them under CurrNode */  
SetSkipListNodeNext (CurrNode, SkipListNodeNext (Brother));  
if (SkipListFreeNode (SkipList, Brother))  
    return (-1);  
return (2);  
}  
  
/* ----- */  
/* SkipListCreate */  
/* ----- */  
/* This function creates a balanced deterministic skip list. */  
/* It returns 0 if everything is OK, and 1 otherwise. */  
/* ----- */  
int SkipListCreate (SkipList, KeyCompare)  
SKIPLIST      *SkipList;  
IntFctPtr     KeyCompare;  
{  
    BLIST          NewList;  
  
    if ((*SkipList = (SKIPLIST)malloc (1, sizeof (SKIPLIST_REC))) == NULL)  
        return (1);  
  
    SetSkipListKeyCompare (*SkipList, KeyCompare);  
  
    /* Creating List of nodes segments */  
    if (BListCreate (&NewList))  
        return (1);  
    SetSkipListNodesSegments (*SkipList, NewList);
```

skiplist.c

```
/* Creating List of used nodes */  
if (BListCreate (&NewList))  
    return (1);  
SetSkipListUsedNodes (*SkipList, NewList);  
  
return (0);  
}  
  
/* ----- */  
/* SkipListQuickDelete */  
/* ----- */  
/* This function frees memory from the whole skip list without freeing */  
/* key and data pointers in each node. */  
/* ----- */  
void SkipListQuickDelete (SkipList)  
    SKIPLIST *SkipList;  
{  
    if (*SkipList == NULL)  
        return;  
  
    BListDelete (&SkipListNodesSegments (*SkipList), SkipListFreeSegment);  
    BListQuickDelete (&SkipListUsedNodes (*SkipList));  
    free (*SkipList);  
    *SkipList = NULL;  
}  
  
/* ----- */  
/* SkipListDelete */  
/* ----- */  
/* This function frees memory from the whole skip list including */  
/* key and data pointers in each node. */  
/* ----- */  
void SkipListDelete (SkipList, KeyDelete, DataDelete)  
    SKIPLIST *SkipList;  
    VoidFctPtr KeyDelete;  
    VoidFctPtr DataDelete;  
{  
    SKIPLIST_NODE CurrNode;  
  
    if (*SkipList == NULL)  
        return;  
  
    if (CurrNode = SkipListRoot (*SkipList))  
    {  
        while (SkipListNodeDown (CurrNode))  
            CurrNode = SkipListNodeDown (CurrNode);  
  
        while (CurrNode)  
        {  
            KeyDelete (SkipListNodeKeyPtr (CurrNode));  
            DataDelete (SkipListNodeDataPtr (CurrNode));  
            CurrNode = SkipListNodeNext (CurrNode);  
        }  
    }  
    SkipListQuickDelete (SkipList);  
}
```

skiplist.c

```
/* ----- */
/* SkipListFindData */          */
/* ----- */
/* This function returns pointer to data, located in the node with a   */
/* key pointed by the given KeyPtr. If the key pointed by the given   */
/* KeyPtr dosn't exist in the skip list, NULL is returned.           */
/* ----- */
DATA_PTR SkipListFindData (SkipList, KeyPtr)
{
    SKIPLIST      SkipList;
    KEY_PTR       KeyPtr;
{
    SKIPLIST_NODE CurrNode;
    IntFctPtr     KeyCompare;

    if (!SkipList)
    {
        fprintf (stderr, "ERROR: The skip list doesn't exist !\n");
        return (NULL);
    }

    CurrNode = SkipListRoot (SkipList);
    KeyCompare = SkipListKeyCompare (SkipList);

    if (!CurrNode || (KeyCompare (KeyPtr, SkipListNodeKeyPtr (CurrNode)) < 0))
        return (NULL);

    while (1)
    {
        while (SkipListNodeNext (CurrNode) &&
               KeyCompare (KeyPtr,
                            SkipListNodeKeyPtr (SkipListNodeNext (CurrNode))) >= 0)
            CurrNode = SkipListNodeNext (CurrNode);

        if (SkipListNodeIsLeaf (CurrNode))
            break;

        CurrNode = SkipListNodeDown (CurrNode);
    }

    if (!KeyCompare (KeyPtr, SkipListNodeKeyPtr (CurrNode)))
        return (SkipListNodeDataPtr (CurrNode));
    else
        return (NULL);
}

/* ----- */
/* SkipListInsert */          */
/* ----- */
/* This function inserts a given pointer to data into the skip list */
/* according to a key, pointed by a given pointer. It returns 0, if */
/* everything is OK, 1 if the node with key, pointed by a given */
/* pointer, is already stored in the skip list; and -1 otherwise. */
/* ----- */
int SkipListInsert (SkipList, KeyPtr, DataPtr)
{
    SKIPLIST      SkipList;
    KEY_PTR       KeyPtr;
```

skiplist.c

```
DATA_PTR           DataPtr;
{
    IntFctPtr      KeyCompare;
    SKIPLIST_NODE   NewNode;
    SKIPLIST_NODE   NewDown;
    int             Status;

    if (!SkipList)
    {
        fprintf (stderr, "ERROR: The skip list doesn't exist !\n");
        return (-1);
    }

    KeyCompare = SkipListKeyCompare (SkipList);

    if (!SkipListRoot (SkipList))
    {
        if (!(NewNode = SkipListCreateNode (SkipList)))
            return (1);
        SetSkipListNodeKeyPtr (NewNode, KeyPtr);
        SetSkipListNodeDataPtr (NewNode, DataPtr);
        SetSkipListRoot (SkipList, NewNode);
        SkipListSizeIncr (SkipList);
        return (0);
    }

    Status = SkipListInsertRec (SkipList, SkipListRoot (SkipList),
                                KeyPtr, DataPtr, &NewDown);

    if (!Status)
        return (0);

    if (Status == -1)
        return (-1);

    if (Status == -2)
        return (1);

    if (!(NewNode = SkipListCreateNode (SkipList)))
        return (1);
    SetSkipListNodeKeyPtr (NewNode, SkipListNodeKeyPtr (NewDown));
    SetSkipListNodeDown (NewNode, NewDown);
    SetSkipListRoot (SkipList, NewNode);
    return (0);
}

/* -----
/* SkipListRemove
/* -----
/* This function removes the node with the key, pointed by the given
/* KeyPtr. If there was something wrong, -1 is returned. If the key,
/* pointed by the given KeyPtr, was in the skip list, 0 is returned
/* and in the parameters NodesKeyPtr and NodesDataPtr the corresponding
/* node's KeyPtr and DataPtr are returned. Otherwise 1 is returned.
/* -----
int SkipListRemove (SkipList, KeyPtr, NodesKeyPtr, NodesDataPtr)
    SKIPLIST          SkipList;
```

skiplist.c

```
KEY_PTR           KeyPtr;
KEY_PTR           *NodesKeyPtr;
DATA_PTR          *NodesDataPtr;
{
    SKIPLIST_NODE      Root;
    SKIPLIST_NODE      NewDown;
    IntFctPtr          KeyCompare;
    int                Status;

    if (!SkipList)
    {
        fprintf (stderr, "ERROR: The skip list doesn't exist !\n");
        return (1);
    }

    Root = SkipListRoot (SkipList);
    KeyCompare = SkipListKeyCompare (SkipList);

    if (!Root)
        return (1);

    if (SkipListNodeIsLeaf (Root))
    {
        if (!KeyCompare (KeyPtr, SkipListNodeKeyPtr (Root)))
        {
            *NodesKeyPtr = SkipListNodeKeyPtr (Root);
            *NodesDataPtr = SkipListNodeDataPtr (Root);
            if (SkipListFreeNode (SkipList, Root))
                return (-1);
            SetSkipListRoot (SkipList, NULL);
            SkipListSizeDecr (SkipList);
            return(0);
        }
        return (1);
    }

    Status = SkipListRemoveRec (SkipList, SkipListNodeDown (Root), NULL,
                                KeyPtr, NodesKeyPtr, NodesDataPtr, &NewDown);

    if (Status != 1)
    {
        if (SkipListNodeDown (Root) != NewDown)
            SetSkipListNodeDown (Root, NewDown);
        SetSkipListNodeKeyPtr (Root,
                               SkipListNodeKeyPtr (SkipListNodeDown (Root)));
    }

    if (Status < 2)
        return (Status);

    if (!SkipListNodeNext (SkipListNodeDown (Root)))
    {
        SetSkipListRoot (SkipList, SkipListNodeDown (Root));
        if (SkipListFreeNode (SkipList, Root))
            return (-1);
    }
}
```

skiplist.c

```
    return (0);
}

/* -----
/* SkipListFirstData
/* -----
/* This function initializes the iterator to point on the first node */
/* of the skip list. It returns pointer to the first data. or NULL, if */
/* the given skip list doesn't exist or is empty. In the second */
/* parameter the corresponding pointer to key is returned. */
/* ----- */

DATA_PTR SkipListFirstData (SkipList, FirstKeyPtr)
    SKIPLIST           SkipList;
    KEY_PTR            *FirstKeyPtr;
{
    if (!SkipList)
    {
        fprintf (stderr, "ERROR: The skip list doesn't exist !\n");
        *FirstKeyPtr = NULL;
        return (NULL);
    }

    if (!SkipListRoot (SkipList))
    {
        SetSkipListCurrNode (SkipList, NULL);
        *FirstKeyPtr = NULL;
        return (NULL);
    }

    SetSkipListCurrNode (SkipList, SkipListRoot (SkipList));

    while (SkipListNodeDown (SkipListCurrNode (SkipList)))
        SetSkipListCurrNode (SkipList,
                            SkipListNodeDown (SkipListCurrNode (SkipList)));

    *FirstKeyPtr = SkipListNodeKeyPtr (SkipListCurrNode (SkipList));
    return (SkipListNodeDataPtr (SkipListCurrNode (SkipList)));
}

/* -----
/* SkipListNextData
/* -----
/* This function returns the next pointer to data, or NULL, if the */
/* iterator reached the end of the skip list. One must call to the */
/* SkipListFirstData function before this one. In the second */
/* parameter the corresponding pointer to key is returned. */
/* ----- */

DATA_PTR SkipListNextData (SkipList, NextKeyPtr)
    SKIPLIST           SkipList;
    KEY_PTR            *NextKeyPtr;
{
    if (!SkipListCurrNode (SkipList))
    {
        *NextKeyPtr = NULL;
        return (NULL);
    }
```

skiplist.c

```
SetSkipListCurrNode (SkipList,
                     SkipListNodeNext (SkipListCurrNode (SkipList)));

if (SkipListCurrNode (SkipList))
{
    *NextKeyPtr = SkipListNodeKeyPtr (SkipListCurrNode (SkipList));
    return (SkipListNodeDataPtr (SkipListCurrNode (SkipList)));
}

*NextKeyPtr = NULL;
return (NULL);
}

/*
 * SkipListPrint
 */
/*
 * This function prints all keys and corresponding data, which stored
 * in the given skip list, in ascending order of the keys.
 * The function receives as parameters pointer to functions that get
 * KeyPtr and DataPtr respectively and print the key and data pointed
 * by them.
 */
void SkipListPrint (SkipList, PrintKey, PrintData)
{
    SKIPLIST           SkipList;
    VoidFctPtr         PrintKey;
    VoidFctPtr         PrintData;
{
    KEY_PTR            KeyPtr;
    DATA_PTR           DataPtr;

    if (SkipList == NULL)
    {
        printf ("The skip list doesn't exist !\n");
        return;
    }

    if (!SkipListRoot (SkipList))
    {
        printf ("The skip list is empty !\n");
        return;
    }

    printf ("The contents of the skip list:\n");

    for (DataPtr = SkipListFirstData (SkipList, &KeyPtr),
         DataPtr;
         DataPtr = SkipListNextData (SkipList, &KeyPtr))
    {
        printf ("    ");
        PrintKey (KeyPtr);
        PrintData (DataPtr);
    }
}
```

skiplist.h

```
/*-----*/
/*
 *      File:          skiplist.h
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Class: none
 *      Inheritance:
 */
/*-----*/
#ifndef SKIPLIST_H
#define SKIPLIST_H

#include "blist.h"
#include "blist.e"

#define SKIPLIST_NODE_SEGMENT_SIZE    100

typedef void *KEY_PTR;
typedef void *DATA_PTR;

/* -----
/* SKIPLIST_NODE_STRUCT
/* ----- */
typedef struct SKIPLIST_NODE_STRUCT {
    KEY_PTR             KeyPtr;
    DATA_PTR            DataPtr;
    struct SKIPLIST_NODE_STRUCT *Next;
    struct SKIPLIST_NODE_STRUCT *Down;
}
SKIPLIST_NODE_REC, *SKIPLIST_NODE;

#define SkipListNodeIsLeaf(n)           (((n)->Down) == NULL)
#define SkipListNodeKeyPtr(n)           (((n)->KeyPtr))
#define SkipListNodeDataPtr(n)          (((n)->DataPtr))
#define SkipListNodeNext(n)             (((n)->Next))
#define SkipListNodeDown(n)             (((n)->Down))

#define SetSkipListNodeKeyPtr(n,k)       (((n)->KeyPtr = k))
#define SetSkipListNodeDataPtr(n,d)     (((n)->DataPtr = d))
#define SetSkipListNodeNext(n,b)        (((n)->Next = b))
#define SetSkipListNodeDown(n,s)        (((n)->Down = s))

/* -----
/* SKIPLIST_STRUCT
/* ----- */
typedef struct SKIPLIST_STRUCT {
    BLIST               NodesSegments; /* Segments of SKIPLIST_NODE */
    BLIST               UsedNodes;      /* allocated nodes,
                                         /* which were removed */
    SKIPLIST_NODE       FirstNode;
    SKIPLIST_NODE       LastNode;
    SKIPLIST_NODE       Root;
    SKIPLIST_NODE       CurrNode;      /* For iterator */
    int                 Size;
    IntFctPtr          KeyCompare;
}
```

skiplist.h

```
}

SKIPLIST_REC, *SKIPLIST;

#define SkipListNodesSegments(s) (((s)->NodesSegments))
#define SkipListUsedNodes(s) (((s)->UsedNodes))
#define SkipListFirstNode(s) (((s)->FirstNode))
#define SkipListLastNode(s) (((s)->LastNode))
#define SkipListRoot(s) (((s)->Root))
#define SkipListCurrNode(s) (((s)->CurrNode))
#define SkipListSize(s) (((s)->Size))
#define SkipListKeyCompare(s) (((s)->KeyCompare))

#define SetSkipListNodesSegments(s,n) (((s)->NodesSegments = n))
#define SetSkipListUsedNodes(s,u) (((s)->UsedNodes = u))
#define SetSkipListFirstNode(s,n) (((s)->FirstNode = n))
#define SetSkipListLastNode(s,n) (((s)->LastNode = n))
#define SetSkipListRoot(s,r) (((s)->Root = r))
#define SetSkipListCurrNode(s,n) (((s)->CurrNode = n))
#define SkipListSizeIncr(s) (((s)->Size++))
#define SkipListSizeDecr(s) (((s)->Size--))
#define SetSkipListSize(s,n) (((s)->Size = n))
#define SetSkipListKeyCompare(s,f) (((s)->KeyCompare = f))

/* -----
 * -----
 * SkipListCreate */ */
/* This function creates a balanced deterministic skip list. */
/* It returns 0 if everything is OK, and 1 otherwise. */
/* -----
 * int SkipListCreate (SKIPLIST*, IntFctPtr); */

/* -----
 * SkipListQuickDelete */ */
/* This function frees memory from the whole skip list without freeing */
/* key and data pointers in each node. */
/* -----
 * void SkipListQuickDelete (SKIPLIST*); */

/* -----
 * SkipListDelete */ */
/* This function frees memory from the whole skip list including */
/* key and data pointers in each node. */
/* -----
 * void SkipListDelete (SKIPLIST*, VoidFctPtr, VoidFctPtr); */

/* -----
 * SkipListFindData */ */
/* This function returns pointer to data, located in the node with a */
/* key pointed by the given KeyPtr. If the key pointed by the given */
/* KeyPtr doesn't exist in the skip list, NULL is returned. */
/* -----
 * DATA_PTR SkipListFindData (SKIPLIST, KEY_PTR);
```

skiplist.h

```
/* ----- */
/* SkipListInsert */ 
/* ----- */
/* This function inserts a given pointer to data into the skip list */
/* according to a key, pointed by a given pointer. It returns 0, if */
/* everything is OK, 1 if the node with key, pointed by a given */
/* pointer, is already stored in the skip list; and -1 otherwise. */
/* ----- */
int SkipListInsert (SKIPLIST, KEY_PTR, DATA_PTR);

/* ----- */
/* SkipListRemove */ 
/* ----- */
/* This function removes the node with the key, pointed by the given */
/* KeyPtr. If there was something wrong, -1 is returned. If the key, */
/* pointed by the given KeyPtr, was in the skip list, 0 is returned */
/* and in the parameters NodesKeyPtr and NodesDataPtr the corresponding */
/* node's KeyPtr and DataPtr are returned. Otherwise 1 is returned. */
/* ----- */
int SkipListRemove (SKIPLIST, KEY_PTR, KEY_PTR*, DATA_PTR*);

/* ----- */
/* SkipListInitIterator */ 
/* ----- */
/* This function initializes the iterator to point on the first node */
/* of the skip list. It returns pointer to the first data. or NULL, if */
/* the given skip list doesn't exist or is empty. In the second */
/* parameter the corresponding pointer to key is returned. */
/* ----- */
DATA_PTR SkipListFirstData (SKIPLIST, KEY_PTR*); 

/* ----- */
/* SkipListNextData */ 
/* ----- */
/* This function returns the next pointer to data, or NULL, if the */
/* iterator reached the end of the skip list. One must call to the */
/* SkipListFirstData function before this one. In the second */
/* parameter the corresponding pointer to key is returned. */
/* ----- */
DATA_PTR SkipListNextData (SKIPLIST, KEY_PTR*); 

/* ----- */
/* SkipListPrint */ 
/* ----- */
/* This function prints all keys and corresponding data, which stored */
/* in the given skip list, in ascending order of the keys. */
/* The function receives as parameters pointer to functions that get */
/* KeyPtr and DataPtr respectively and print the key and data pointed */
/* by them. */
/* ----- */
void SkipListPrint (SKIPLIST, VoidFctPtr, VoidFctPtr);

/*----- EOF -----*/
#endif
```

time.h

```
/*
 * File:          time.h
 * Version:       1.1
 * Modifications: -
 * Documentation: -
 */
/*
 */
/*
 */
/*-----*/
/* Timing Info attached to GNL_VAR */
/*-----*/
typedef struct GNL_TIMING_INFO_STRUCT
{
    float           ArrivalTimeFall;
    float           ArrivalTimeRise;
    float           RequiredTimeFall;
    float           RequiredTimeRise;
    float           TransitionTimeFall;
    float           TransitionTimeRise;
    float           OutCapacitance;
    int             OutFanout;
    int             Tag;
    void            *Hook;
}
GNL_TIMING_INFO_REC, *GNL_TIMING_INFO;

#define GnlTimingInfoArrivalFall(n) ((n)->ArrivalTimeFall)
#define GnlTimingInfoArrivalRise(n) ((n)->ArrivalTimeRise)
#define GnlTimingInfoRequiredFall(n) ((n)->RequiredTimeFall)
#define GnlTimingInfoRequiredRise(n) ((n)->RequiredTimeRise)
#define GnlTimingInfoFanout(n) ((n)->OutFanout)
#define GnlTimingInfoCapa(n) ((n)->OutCapacitance)
#define GnlTimingInfoTransitionFall(n) ((n)->TransitionTimeFall)
#define GnlTimingInfoTransitionRise(n) ((n)->TransitionTimeRise)
#define GnlTimingInfoTag(n) ((n)->Tag)
#define GnlTimingInfoHook(n) ((n)->Hook)

#define SetGnlTimingInfoArrivalFall(n, b) ((n)->ArrivalTimeFall = b)
#define SetGnlTimingInfoArrivalRise(n, b) ((n)->ArrivalTimeRise = b)
#define SetGnlTimingInfoRequiredFall(n, b) ((n)->RequiredTimeFall = b)
#define SetGnlTimingInfoRequiredRise(n, b) ((n)->RequiredTimeRise = b)
#define SetGnlTimingInfoFanout(n, b) ((n)->OutFanout = b)
#define SetGnlTimingInfoCapa(n, b) ((n)->OutCapacitance = b)
#define SetGnlTimingInfoTransitionFall(n, b) ((n)->TransitionTimeFall = b)
#define SetGnlTimingInfoTransitionRise(n, b) ((n)->TransitionTimeRise = b)
#define SetGnlTimingInfoTag(n, b) ((n)->Tag = b)
#define SetGnlTimingInfoHook(n, b) ((n)->Hook = b)

/*-----*/
typedef enum {
    PORT_IN,
    PORT_OUT,
    PORT_INOUT,
    PORT_LOCAL
} GNL_PORT_DIR;
```

```

/*-----*/
/* GNL_CRITICAL_PATH */
/*-----*/
typedef struct GNL_CRITICAL_PATH_STRUCT
{
    GNL_VAR          CriticalVar;           /* Critical Var on*/
    GNL_ASSOC        CriticalAssoc;
    char             *InstName;
    int              Fanout;
    float            Incr;
    float            ArrivalTime;
    float            RequiredTime;
    int              KindOfTime;
    int              Tag;
    GNL_PORT_DIR    Direction;
    BLIST            Predecessors;         /* GNL_CRITICAL_PATH */
    BLIST            Successors;          /* GNL_CRITICAL_PATH */
}
GNL_CRITICAL_PATH_REC, *GNL_CRITICAL_PATH;

#define GnlCritPathCriticalVar(n)   ((n)->CriticalVar)
#define GnlCritPathCriticalAssoc(n) ((n)->CriticalAssoc)
#define GnlCritPathInstName(n)      ((n)->InstName)
#define GnlCritPathFanout(n)       ((n)->Fanout)
#define GnlCritPathIncr(n)        ((n)->Incr)
#define GnlCritPathArrTime(n)     ((n)->ArrivalTime)
#define GnlCritPathReqTime(n)     ((n)->RequiredTime)
#define GnlCritPathKindOfTime(n)   ((n)->KindOfTime)
#define GnlCritPathTag(n)         ((n)->Tag)
#define GnlCritPathPredecessors(n) ((n)->Predecessors)
#define GnlCritPathSuccessors(n)   ((n)->Successors)

#define SetGnlCritPathCriticalVar(n, b)          ((n)->CriticalVar = b)
#define SetGnlCritPathCriticalAssoc(n, b)         ((n)->CriticalAssoc = b)
#define SetGnlCritPathInstName(n, b)              ((n)->InstName = b)
#define SetGnlCritPathFanout(n, b)                ((n)->Fanout = b)
#define SetGnlCritPathIncr(n, b)                  ((n)->Incr = b)
#define SetGnlCritPathArrTime(n, b)               ((n)->ArrivalTime = b)
#define SetGnlCritPathReqTime(n, b)               ((n)->RequiredTime = b)
#define SetGnlCritPathKindOfTime(n, b)            ((n)->KindOfTime = b)
#define SetGnlCritPathTag(n, b)                   ((n)->Tag = b)
#define SetGnlCritPathPredecessors(n, b)          ((n)->Predecessors = b)
#define SetGnlCritPathSuccessors(n, b)            ((n)->Successors = b)

/*-----*/
/* GNL_PATH_ELEM */
/*-----*/
typedef struct GNL_PATH_ELEM_STRUCT
{
    char             *InPort;
    char             *OutPort;
    GNL_VAR          CriticalVar;
    GNL_COMPONENT   Compo;
    char             *InstName;
    int              Fanout;
    float            RiseIncr;
}

```

```

float          FallIncr;
float          RiseArrivalTime;
float          FallArrivalTime;
float          RiseRequiredTime;
float          FallRequiredTime;
}

GNL_PATH_ELEM_REC, *GNL_PATH_ELEM;

#define GnlPathElemInPort(n)          ((n)->InPort)
#define GnlPathElemOutPort(n)         ((n)->OutPort)
#define GnlPathElemCompo(n)          ((n)->Compo)
#define GnlPathElemFanout(n)         ((n)->Fanout)
#define GnlPathElemCriticalVar(n)    ((n)->CriticalVar)
#define GnlPathElemInstName(n)       ((n)->InstName)
#define GnlPathElemRiseArrTime(n)    ((n)->RiseArrivalTime)
#define GnlPathElemFallArrTime(n)    ((n)->FallArrivalTime)
#define GnlPathElemRiseReqTime(n)    ((n)->RiseRequiredTime)
#define GnlPathElemFallReqTime(n)    ((n)->FallRequiredTime)
#define GnlPathElemRiseIncr(n)       ((n)->RiseIncr)
#define GnlPathElemFallIncr(n)       ((n)->FallIncr)

#define SetGnlPathElemInPort(n, b)   ((n)->InPort = b)
#define SetGnlPathElemOutPort(n, b)  ((n)->OutPort = b)
#define SetGnlPathElemCompo(n, b)   ((n)->Compo = b)
#define SetGnlPathElemFanout(n, b)  ((n)->Fanout = b)
#define SetGnlPathElemCriticalVar(n, b) ((n)->CriticalVar = b)
#define SetGnlPathElemInstName(n, b) ((n)->InstName = b)
#define SetGnlPathElemRiseArrTime(n, b) ((n)->RiseArrivalTime = b)
#define SetGnlPathElemFallArrTime(n, b) ((n)->FallArrivalTime = b)
#define SetGnlPathElemRiseReqTime(n, b) ((n)->RiseRequiredTime = b)
#define SetGnlPathElemFallReqTime(n, b) ((n)->FallRequiredTime = b)
#define SetGnlPathElemRiseIncr(n, b) ((n)->RiseIncr = b)
#define SetGnlPathElemFallIncr(n, b) ((n)->FallIncr = b)

/*-----*/
/* GNL_CLOCK
/*-----*/
typedef struct GNL_CLOCK_STRUCT
{
    GNL_VAR        SourceClock;
    char           *HierNameOfGnlClok;
    float          ArrivalTime;
    float          RequiredTime;
    float          SetupTime;
    BLIST          SequentialInstances; /* GNL_SEQUENTIAL_COMPONENT */
    BLIST          SeqhierInstNames; /* char */
                                /* SeqhierInstNames[i] is the global
                                   name including the hierarchy, it
                                   corresponds to SequentialInstances[i] */
}
GNL_CLOCK_REC, *GNL_CLOCK;

#define GnlClockSourceClock(n)          ((n)->SourceClock)
#define GnlClockHierNameOfGnlClok(n)    ((n)->HierNameOfGnlClok)
#define GnlClockArrTime(n)             ((n)->ArrivalTime)
#define GnlClockReqTime(n)             ((n)->RequiredTime)
#define GnlClockSetupTime(n)           ((n)->SetupTime)

```

time.h

```
#define GnlClockSeqInst(n)          ((n)->SequentialInstances)
#define GnlClockSeqHierInstNames(n)  ((n)->SeqhierInstNames)

#define SetGnlClockSourceClock(n, b)           ((n)->SourceClock = b)
#define SetGnlClockHierNameOfGnlClock(n, b)    ((n)->HierNameOfGnlClock = b)
#define SetGnlClockArrTime(n, b)               ((n)->ArrivalTime = b)
#define SetGnlClockReqTime(n, b)               ((n)->RequiredTime = b)
#define SetGnlClockSetupTime(n, b)              ((n)->SetupTime = b)
#define SetGnlClockSeqInst(n, b)               ((n)->SequentialInstances = b)
#define SetGnlClockSeqHierInstNames(n, b)      ((n)->SeqhierInstNames = b)

/*-----*/
/* TIME_SEQUENTIAL_ELEM */
/*-----*/
typedef struct TIME_SEQUENTIAL_ELEM_STRUCT
{
    GNL_SEQUENTIAL_COMPONENT SeqComp;
    BLIST HierInstPath;
    float SlackAtOutput;
}
TIME_SEQUENTIAL_ELEM_REC, *TIME_SEQUENTIAL_ELEM;

#define TimeGetSeqElemSeqComp(n)          ((n)->SeqComp)
#define TimeGetSeqElemHierInstPath(n)     ((n)->HierInstPath)
#define TimeGetSeqElemSlackAtOutput(n)   ((n)->SlackAtOutput)

#define SetTimeGetSeqElemSeqComp(n, b)    ((n)->SeqComp = b)
#define SetTimeGetSeqElemHierInstPath(n, b) ((n)->HierInstPath = b)
#define SetTimeGetSeqElemSlackAtOutput(n, b) ((n)->SlackAtOutput = b)

/*-----*/
#define RISE      'R'
#define FALL      'F'
/*-----*/
/*----- Global Variable of Traversal of the Netlist -----*/
extern GNL_USER_COMPONENT      G_CurrentComponent;
extern BLIST                   G_PileOfComponent;

/*-----*/
#define MAX_FLOAT 9999
/*-----*/
/* ----- EOF ----- */
/*-----*/
```

timecomp.c

```
/*-----*/  
/*-----*/  
/*      File:          timecomp.c */  
/*      Version:        1.1 */  
/*      Modifications: - */  
/*      Documentation: - */  
/*-----*/  
/*      Description: */  
/*-----*/  
  
#include <stdio.h>  
  
#ifdef MEMORY           /* If one wants memory statistics for SUN */  
#include <malloc.h>  
#endif  
  
#include "blist.h"  
#include "gnl.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnlestim.h"  
#include "time.h"  
#include "bbdd.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
#include "gnloption.h"  
#include "gnllibc.h"  
  
#include "blist.e"  
#include "libutil.e"  
#include "timeutil.e"  
#include "timecomp.e"  
#include "timepath.e"  
#include "timefan.e"  
  
#define MAX_CELL_NUMBER      30000  
/*-----*/  
/* EXTERN */  
/*-----*/  
extern GNL_ENV      G_GnlEnv;  
/*-----*/  
  
/*-----*/  
/*----- Global Variable of Traversal of the Netlist -----*/  
  
GNL_USER_COMPONENT      G_CurrentComponent;  
BLIST                  G_PileOfComponent;  
/*-----*/  
  
/*-----*/  
int GnlFloatEqual (float a, float b, int precision)  
{  
    float f, puiss = 1.0;  
    int i;
```

timecomp.c

```
for (i=0 ; i< precision ; i++)
    puiss *= 10;

if (a > b)
    f = (a - b) * puiss ;
else
    f = (b - a) * puiss;

i = (f < 1) ? 1 : 0;
return (i);
}

/*-----*/
float GnlMaxFloat (float a, float b)
{
    float Max;

    Max = a;
    if (Max < b)
        Max = b;

    return (Max);
}

/*-----*/
float GnlMinFloat (float a, float b)
{
    float Min;

    Min = a;
    if (Min > b)
        Min = b;

    return (Min);
}

/*-----*/
/*-----*/
/* GnlGetHierarchycalInstName */
/*-----*/
GNL_STATUS GnlGetHierarchycalInstName (char *InstanceName,
                                         char **HierarchycalName)
{
    char *Str1, *Str2, *InstName;
    int i;
    GNL_USER_COMPONENT      AuxUserCompo;

    InstName = NULL;
    if (BListSize (G_PileOfComponent))
    {
        AuxUserCompo = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent, 0);
        Str2 = GnlUserComponentInstName (AuxUserCompo);
        if (GnlStrCopy (Str2, &InstName))
            return (GNL_MEMORY_FULL);
        for (i=1; i < BListSize (G_PileOfComponent); i++)
        {
            Str1 = InstName;
```

timecomp.c

```
if (GnlStrAppendStrCopy (Str1, GnlEnvFlattenStr(), &InstName))
    return (GNL_MEMORY_FULL);
free (Str1);
Str1 = InstName;
AuxUserCompo = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent, i);
Str2 = GnlUserComponentInstName (AuxUserCompo);
if (GnlStrAppendStrCopy (Str1, Str2, &InstName))
    return (GNL_MEMORY_FULL);
free (Str1);
}
if (InstanceName)
{
    Str1 = InstName;
    if (GnlStrAppendStrCopy (Str1, GnlEnvFlattenStr(), &InstName))
        return (GNL_MEMORY_FULL);
    free (Str1);
    Str1 = InstName;
    if (GnlStrAppendStrCopy (Str1, InstanceName, &InstName))
        return (GNL_MEMORY_FULL);
    free (Str1);
}
else
if (InstanceName)
    if (GnlStrCopy (InstanceName, &InstName))
        return (GNL_MEMORY_FULL);
*HierarchycalName = InstName;
return (GNL_OK);
}

/*-----
/* SetGnlCritPathInstNameFromUserCompo
*/
GNL_STATUS SetGnlCritPathInstNameFromUserCompo (GNL_CRITICAL_PATH CritPath,
                                                char *InstanceName)
{
    char *HierarchycalName;

    if (!CritPath || !InstanceName)
        return (GNL_OK);
    if (GnlGetHierarchycalInstName (InstanceName, &HierarchycalName))
        return (GNL_MEMORY_FULL);

    SetGnlCritPathInstName (CritPath, HierarchycalName);
    return (GNL_OK);
}
/*-----
/* GnlPrintCritPath
*/
void GnlPrintCritPath (BLIST ListCritPath, float *ClockFreqValue,
                      float *CombPathValue)
{
    GNL_CRITICAL_PATH      CritPath;
    LIBC_CELL              Cell;
    GNL_COMPONENT          Compo;
    GNL_VAR                Var, Formal, StartVar, EndVar;
    int                   i, IsSequential;
```

timecomp.c

```

GNL_ASSOC          Assoc;

CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath,
                                         BListSize(ListCritPath) -1);
StartVar = GnlCritPathCriticalVar (CritPath);
CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, 0);
EndVar = GnlCritPathCriticalVar (CritPath);

IsSequential = (!StartVar || !EndVar);

fprintf (stderr, " Point\t\t\t\t\tFanout\tIncr\tPath\n");
fprintf (stderr, " -----");
-----\n");

for (i= BListSize(ListCritPath) -1; i >= 0; i--)
{
    CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, i);
    if (!GnlCritPathPredecessors (CritPath) ||
        !GnlCritPathSuccessors (CritPath))
    {
        Var = GnlCritPathCriticalVar (CritPath);
        if (Var)
        {
            if (!GnlCritPathPredecessors (CritPath))
            {
                /* the input external delay is depending of constraint timing */
                /* For the moment it is 0.0 (later we have to do that      */
                fprintf (stderr, " input external delay\t\t\tt0.0\tt0.0 \n");
            }
            fprintf (stderr, " %s ", GnlVarName (Var));

            if (!GnlCritPathPredecessors (CritPath))
                fprintf (stderr, "(in)\n\t\t\t\t\t\t\t\t");
            else
                fprintf (stderr, "(out)\n\t\t\t\t\t\t\t\t\t");
            fprintf (stderr, "% .2f\t% .2f", GnlCritPathIncr (CritPath),
                     GnlCritPathArrTime (CritPath));
            if (GnlCritPathKindOfTime (CritPath) == RISE)
                fprintf (stderr, " Rising\n");
            else
                fprintf (stderr, " Falling\n");
        }
    }
}

Assoc = GnlCritPathCriticalAssoc (CritPath);
if (Assoc)
{
    Compo = GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);
    Formal = GnlAssocFormalPort (Assoc);
    GnlGetCellFromGnlCompo (Compo, &Cell);
    if (Cell)
    {
        fprintf (stderr, " %s", GnlCritPathInstName (CritPath));
        fprintf (stderr, "/%s %s", (char *) Formal, GnlVarName (Var));
        fprintf (stderr, " (%s)\n\t\t\t\t\t\t\t", LibCellName (Cell));
    }
}

```

timecomp.c

timecomp.c

```
if (!CritPath || !ListCritPath)
    return (GNL_OK);

if (NumberOfPaths <= 0)
    return (GNL_OK);

if (BListAddElt (ListCritPath, CritPath))
    return (GNL_MEMORY_FULL);

if (GnlCritPathPredecessors (CritPath))
{
    for (i=0; i < BListSize (GnlCritPathPredecessors (CritPath)); i++)
    {
        Predecessor = (GNL_CRITICAL_PATH) BListElt (
            GnlCritPathPredecessors (CritPath), i);

        GnlPrintNEquivPathFromOneCritPath (Predecessor, ListCritPath,
            NumberofPaths, IsSequential, ClockFreqValue,
            CombPathValue, MinSlackForSeqPath,
            MinSlackForCombPath,
            ListPathClockFreq);
        BListDelShift (ListCritPath, BListSize (ListCritPath));
    }
}
else
{
    if (*NumberOfPaths > 0)
    {
        EndCritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, 0);
        Slack = GnlCritPathReqTime(EndCritPath) -
        GnlCritPathArrTime(EndCritPath);
        StartVar = GnlCritPathCriticalVar (CritPath);
        EndVar = GnlCritPathCriticalVar (EndCritPath);
        if (!IsSequential)
        {
            if (StartVar && GnlFloatEqual (MinSlackForCombPath, Slack, 4))
            {
                GnlPrintCritPath (ListCritPath, ClockFreqValue, CombPathValue);
                (*NumberOfPaths)--;
            }
            else
            {
                if (GnlFloatEqual (MinSlackForSeqPath, Slack, 4))
                    if (BListAddElt (ListPathClockFreq, BListElt (ListCritPath, 0)))
                        return (GNL_MEMORY_FULL);
            }
            return (GNL_OK);
        }
        else
        {
            if (!StartVar || !EndVar)
            {
                GnlPrintCritPath (ListCritPath, ClockFreqValue, CombPathValue);
                (*NumberOfPaths)--;
            }
        }
    }
}
```

timecomp.c

```
    }
    return (GNL_OK);
}

/*-----
/* GnlComparePathToCurrentPath
 *-----
int GnlComparePathToCurrentPath (GNL_PATH_COMPONENT Path,
                                BLIST CurrentPath)
{
    int i;
    GNL_PATH_COMPONENT Previous;
    GNL_USER_COMPONENT CompoI;

    if (!Path)
        return (0);

    CompoI = GnlPathComponentComponent (Path);
    Previous = GnlPathComponentPrevious (Path);
    for (i= BListSize (CurrentPath)-1; i >=0; i--)
    {
        if (CompoI != (GNL_USER_COMPONENT) BListElt (CurrentPath, i))
            return (0);
        if (Previous && (i == 0))
            return (0);
        if (!Previous && (i > 0))
            return (0);
        if (Previous)
        {
            CompoI = GnlPathComponentComponent (Previous);
            Previous = GnlPathComponentPrevious (Previous);
        }
    }

    return (1);
}
/*-----
/* TimingInfoCorrespToCurrentPathFromVar ()
 *-----
GNL_STATUS GnlTimingInfoCorrespToCurrentPathFromVar (
    GNL_VAR Var,
    GNL_TIMING_INFO *TimingInfo,
    unsigned int *Key,
    int *Rank)
{
    GNL_STATUS GnlStatus;
    BLIST ListTimingInfo, ListPaths;
    GNL CurrentGnl;
    GNL_USER_COMPONENT CurrentComponent;
    char *InstanceName;
    BLIST SubList, SubListForTiming;

    *Rank = 0;
    if (BListSize (G_PileOfComponent))
```

```

{
    /* we take the last element of the pile                      */
    /* e.g the current instance.                                */
    CurrentComponent = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                       BListSize (G_PileOfComponent)-1);
    CurrentGnl = GnlUserComponentGnlDef (CurrentComponent);

    /* we access the hash list.                                 */
    ListPaths = GnlListPathComponent (CurrentGnl);
    InstanceName = GnlUserComponentInstName (CurrentComponent);
    *Key = KeyOfName (InstanceName, BListSize (ListPaths));
    SubList = (BLIST)BListElt (ListPaths, *Key);

    *Rank = BListMemberOfList (SubList, G_PileOfComponent,
                             GnlComparePathToCurrentPath);

    ListTimingInfo = (BLIST) GnlVarTraversalInfoHook (Var);
    if (!ListTimingInfo)
    {
        if (BListCreateWithSize (BListSize (ListPaths), &ListTimingInfo))
            return (GNL_MEMORY_FULL);
        BSize (ListTimingInfo) = BListSize (ListPaths);
        SetGnlVarTraversalInfoHook (Var, (void*) ListTimingInfo);

        if (BListCreateWithSize (BListSize (SubList), &SubListForTiming))
            return (GNL_MEMORY_FULL);
        BSize (SubListForTiming) = BListSize (SubList);
        BListElt (ListTimingInfo, *Key) = (int) SubListForTiming;
        *TimingInfo = (GNL_TIMING_INFO) NULL;
    }
    else
    {
        SubListForTiming = (BLIST) BListElt (ListTimingInfo, *Key);
        if (!SubListForTiming)
        {
            if (BListCreateWithSize (BListSize (SubList), &SubListForTiming))
                return (GNL_MEMORY_FULL);
            BSize (SubListForTiming) = BListSize (SubList);
            BListElt (ListTimingInfo, *Key) = (int) SubListForTiming;
        }
        *TimingInfo = (GNL_TIMING_INFO) BListElt (SubListForTiming, *Rank-1);
    }
}
else
    *TimingInfo = (GNL_TIMING_INFO) GnlVarTraversalInfoHook (Var);

return (GNL_OK);
}

/*-----*/
/* GnlComputeCapaAndFanoutFromSeqCell                         */
/*-----*/
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromSeqCell :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the output capacitance "Capa" and the fanout "Fanout" of
       the net "GnlAssocActualPort (Assoc)".
   - it does not include the wire capacitance.
*/

```

timecomp.c

```

    - the GNL_ASSOC is a connector of a Sequential component.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromSeqCell (GNL_ASSOC      Assoc,
                                                float        *Capa,
                                                int         *Fanout,
                                                LIBC_LIB Lib)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinIn;
    GNL_ASSOC     AssocI;
    char          *Formal;
    LIBC_KFATOR   KF;
    float         AuxCapa;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E )
        return (GNL_OK);

    AuxCapa = LibPinCapa (PinIn);
    if (AuxCapa == 0.0)
        AuxCapa = LibDefaultInputPinCap (Lib);
    KF = LibKFactor (Lib);
    AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                           LibKFactorPinCap(KF) [1],
                           LibKFactorPinCap(KF) [2]);
    *Capa += AuxCapa;
    (*Fanout)++;
    return (GNL_OK);

}
/*-----*/
/* GnlComputeCapaAndFanoutFrom3State */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFrom3State :
   - For a given GNL_ASSOC (Assoc) :
      - Compute the output capacitance "Capa" and the fanout "Fanout" of
        the net "GnlAssocActualPort (Assoc)".
      - it does not include the wire capacitance.
      - the GNL_ASSOC is a connector of a 3state component.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFrom3State (GNL_ASSOC      Assoc,
                                                float        *Capa,
                                                int         *Fanout,
                                                LIBC_LIB Lib)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinIn;
    GNL_ASSOC     AssocI;

```

timecomp.c

```
char *Formal;
LIBC_KFATOR KF;
float AuxCapa;
GNL_TRISTATE_COMPONENT TriStateCompo;

TriStateCompo =
(GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent (Assoc) ;
Cell = GnlTriStateCompoInfoCell (TriStateCompo) ;
Formal = (char *) GnlAssocFormalPort (Assoc) ;

if (! (PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E )
    return (GNL_OK);

AuxCapa = LibPinCapa (PinIn);
if (AuxCapa == 0.0)
    AuxCapa = LibDefaultInputPinCap (Lib);
KF = LibKFactor (Lib);
AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                        LibKFactorPinCap(KF) [1],
                        LibKFactorPinCap(KF) [2]) ;

*Capa += AuxCapa;
(*Fanout)++;
return (GNL_OK);

}

/*-----*/
GNL_STATUS TimeGetFormalFromAssocAndActual (GNL_ASSOC Assoc, GNL_VAR Actual,
                                             GNL_VAR *Formal)
{
    GNL_VAR AuxActual, AuxFormal, IndexFormalVar, SplitActualJ;
    GNL_USER_COMPONENT UserCompo;
    GNL GnlDefCompo;
    char *IndexFormalName;
    int i, j, Index, LeftFormIndex, RightFormIndex;
    BLIST ListSplitActuals;
    GNL_STATUS GnlStatus;

*Formal = (GNL_VAR) NULL;
UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc) ;
GnlDefCompo = GnlUserComponentGnlDef (UserCompo);

AuxActual = GnlAssocActualPort (Assoc) ;
AuxFormal = GnlAssocFormalPort (Assoc) ;

if (GnlVarIsVar (AuxActual))
{
    *Formal = AuxFormal;
    return (GNL_OK);
}
else
{
    ListSplitActuals = GnlNodeSons ((GNL_NODE) AuxActual);
    LeftFormIndex = GnlVarMsb (AuxFormal);
    RightFormIndex = GnlVarLsb (AuxFormal) ;
```

```

if (LeftFormIndex < RightFormIndex)
    Index = LeftFormIndex-1;
else
    Index = LeftFormIndex+1;

for (j=0; j<BListSize (ListSplitActuals); j++)
{
    if (LeftFormIndex < RightFormIndex)
        Index++;
    else
        Index--;
SplitActualJ = (GNL_VAR) BListElt (ListSplitActuals, j);
if (SplitActualJ == Actual)
{
    if (GnlVarIndexName (AuxFormal, Index, &IndexFormalName))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlGetVarFromName (GnlDefCompo,
                                         IndexFormalName, &IndexFormalVar)))
    {
        free (IndexFormalName);
        if (GnlStatus == GNL_VAR_NOT_EXISTS)
        {
            fprintf (stderr, " ERROR: cannot find var\n");
            return (GNL_VAR_NOT_EXISTS);
        }
        return (GNL_MEMORY_FULL);
    }
    *Formal = IndexFormalVar;
    return (GNL_OK);
}
}

return (GNL_OK);
}

/*
 * -----
 * GnlComputeCapaAndFanoutFromUserCompo
 * -----
 */
/* Doc procedure GnlComputeCapaAndFanoutFromUserCompo :
 - For a given GNL_ASSOC (Assoc) :
   - Compute the output capacitance "Capa" and the fanout "Fanout" of
     the net "GnlAssocActualPort (Assoc)".
 - it does not include the wire capacitance.
 - the GNL_ASSOC is a connector of a UserComponent
*/
/*
 */
GNL_STATUS GnlComputeCapaAndFanoutFromUserCompo (GNL_ASSOC Assoc,
                                                 float      *Capa,
                                                 int       *Fanout,
                                                 LIBC_LIB Lib,
                                                 GNL_VAR  Actual,
                                                 int       Tag)
{
    GNL_VAR      Var, Formal, VarI, AuxFormal;

```

timecomp.c

```
GNL_USER_COMPONENT      UserCompo;
GNL_TIMING_INFO TimingInfo;
LIBC_CELL               Cell;
LIBC_PIN                Pin;
GNL_STATUS              GnlStatus;
LIBC_KFATOR             KF;
float                  AuxCapa;
int                     Rank, i;
GNL_ASSOC               AssocI;
BLIST                   Interface;
unsigned int             Key;
int                     AuxVarIsInout;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Var = GnlAssocActualPort (Assoc);
Interface = GnlUserComponentInterface (UserCompo);
Formal = GnlAssocFormalPort (Assoc);
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
        if (!(Pin = LibGetPinFromNameAndCell (Cell, (char *) Formal)))
            return (GNL_OK);
        if (LibPinDirection(Pin) == OUTPUT_E)
            return (GNL_OK);

        AuxCapa = LibPinCapa (Pin);
        if (AuxCapa == 0.0)
            AuxCapa = LibDefaultInputPinCap (Lib);
        KF = LibKFactor (Lib);
        AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                               LibKFactorPinCap(KF) [1],
                               LibKFactorPinCap(KF) [2]);
        *Capa += AuxCapa;
        (*Fanout)++;
        return (GNL_OK);
    }
    else
        /* It is a black box without any definition so we stop */
        /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }

    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);
```

timecomp.c

```
if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
    return (GnlStatus);

AuxVarIsInout = (GnlVarDir (Actual) == GNL_VAR_INOUT);

if (GnlComputeCapaAndFanoutFromVar (AuxFormal, Lib, Tag, 1,
AuxVarIsInout))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal,
                                              &TimingInfo, &Key, &Rank))
    return (GNL_MEMORY_FULL);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*Capa += GnlTimingInfoCapa (TimingInfo);
*Fanout += GnlTimingInfoFanout (TimingInfo);

return (GNL_OK);
}

/*
 * GnlComputeCapaAndFanoutFromAssoc
 */
/*
 * Doc procedure GnlComputeCapaAndFanoutFromAssoc :
 * - For a given GNL_ASSOC (Assoc) :
 *   - Compute the output capacitance "Capa" and the fanout "Fanout" of
 *     the net "GnlAssocActualPort (Assoc)".
 *   - it does not include the wire capacitance.
 *   - the GNL_ASSOC is a connector of a (user component "hierrachycal block or
 * generic cell", Sequential component "Flops or Latches", TriStates).
 */
/*
 * GNL_STATUS GnlComputeCapaAndFanoutFromAssoc (GNL_ASSOC          Assoc,
 *                                               float            *Capa,
 *                                               int              *Fanout,
 *                                               LIBC_LIB         Lib,
 *                                               GNL_VAR          Actual,
 *                                               int              Tag)
{
    GNL_STATUS          GnlStatus;
    int                Rank;
    GNL_COMPONENT       GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            if (GnlStatus = GnlComputeCapaAndFanoutFromSeqCell (Assoc, Capa,
                                                               Fanout, Lib))
                return (GnlStatus);
            break;

        case GNL_USER_COMPO:
            if (GnlComputeCapaAndFanoutFromUserCompo (Assoc, Capa, Fanout,
                                                       Lib, Actual, Tag))
                return (GnlStatus);
            break;
    }
}
```

```

        return (GNL_MEMORY_FULL);
    break;

case GNL_TRISTATE_COMPO:
    if (GnlStatus = GnlComputeCapaAndFanoutFrom3State (Assoc,
                                                       Capa, Fanout, Lib))
        return (GnlStatus);
    break;

case GNL_MACRO_COMPO:
    break;

default:
    GnlError (12 /* Unknown component */);
    break;
}

return (GNL_OK);
}
/*-----*/
/* GnlGetVarFromFormalPortAndHierBlock */
/*-----*/
GNL_ASSOC GnlGetAssocFromFormalPortAndHierBlock (GNL_VAR Formal,
                                                GNL_USER_COMPONENT HierBlock,
                                                GNL_VAR *Actual)
{
    BLIST           Interface, ListSplitActuals;
    GNL_VAR     AuxFormal, AuxActual, SplitActualJ, IndexFormalVar;
    int          i, j, LeftFormIndex, RightFormIndex, Index;
    GNL_ASSOC AssocI;
    char         *IndexFormalName;
    GNL_STATUS   GnlStatus;

    *Actual = (GNL_VAR) NULL;
    if (!Formal || !HierBlock)
        return ((GNL_ASSOC) NULL);

    if (!GnlUserComponentGnlDef (HierBlock))
        return ((GNL_ASSOC) NULL);

    Interface = GnlUserComponentInterface (HierBlock);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        AuxFormal = GnlAssocFormalPort (AssocI);
        AuxActual = GnlAssocActualPort (AssocI);
        if (Formal == AuxFormal)
        {
            *Actual = AuxActual;
            if (!GnlVarIsVar (AuxActual))
                return ((GNL_ASSOC) NULL);
            return (AssocI);
        }

        if (!GnlVarIsVar (AuxActual))
        {
            LeftFormIndex = GnlVarMsb (AuxFormal);

```

```

    RightFormIndex = GnlVarLsb (AuxFormal);

    if (LeftFormIndex < RightFormIndex)
        Index = LeftFormIndex-1;
    else
        Index = LeftFormIndex+1;

    ListSplitActuals = GnlNodeSons ((GNL_NODE) AuxActual);
    for (j=0; j < BListSize (ListSplitActuals); j++)
    {
        if (LeftFormIndex < RightFormIndex)
            Index++;
        else
            Index--;
        SplitActualJ = (GNL_VAR) BListElt (ListSplitActuals, j);
        GnlVarIndexName (AuxFormal, Index, &IndexFormalName);
        if ((GnlStatus
=GnlGetVarFromName (GnlUserComponentGnlDef (HierBlock),
                    IndexFormalName, &IndexFormalVar)))
        {
            if (GnlStatus == GNL_VAR_NOT_EXISTS)
            {
                free (IndexFormalName);
                continue;
            }
            return ((GNL_ASSOC) NULL);
        }
        free (IndexFormalName);

        if (Formal == IndexFormalVar)
        {
            *Actual = SplitActualJ;
            return (AssocI);
        }
    }
    return ((GNL_ASSOC) NULL);
}
/*-----*/
/* GnlComputeCapaAndFanoutFromVar */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromVar :
   - For a given GNL_VAR (Var) :
     - Compute the output capacitance and the fanout of
       the net "Var".
     - it does not include the wire capacitance.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromVar (GNL_VAR      Var,
                                             LIBC_LIB     Lib,
                                             int          Tag,
                                             int          InoutIsIn,
                                             int          VarIsInout)
{
    GNL_VAR          AuxVar, Actual;
    GNL_ASSOC        AuxAssocVar;

```

timecomp.c

```
int          i, Fanout, Rank;
GNL_STATUS   GnlStatus;
BLIST        AssocDest, LeftVarAssigneds, ListTimingInfo,
SubList;
GNL_TIMING_INFO TimingInfo, AuxTimingInfo;
GNL_USER_COMPONENT InstOfGnl;
float         Capa;
unsigned int   Key;
int           AuxVarIsInout;

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                           &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if(TimingInfo)
{
    if ((GnlVarDir (Var) == GNL_VAR_INOUT) && InoutIsIn)
    {
        AuxTimingInfo = GnlTimingInfoHook (TimingInfo);
        if (!AuxTimingInfo)
        {
            if (GnlCreateTimingInfo (&AuxTimingInfo))
                return (GNL_MEMORY_FULL);
            SetGnlTimingInfoHook (TimingInfo, AuxTimingInfo);
        }
        else
            return (GNL_OK);
        TimingInfo = GnlTimingInfoHook (TimingInfo);
    }
}
else
{
    if (GnlCreateTimingInfo (&TimingInfo))
        return (GNL_MEMORY_FULL);

    if (!BListSize (G_PileOfComponent))
        SetGnlVarTraversalInfoHook (Var, (void*) TimingInfo);
    else
    {
        ListTimingInfo = (BLIST) GnlVarTraversalInfoHook (Var);
        SubList = (BLIST) BListElt (ListTimingInfo, Key);

        BListElt (SubList, Rank-1) = (int) TimingInfo;
    }
    if ((GnlVarDir (Var) == GNL_VAR_INOUT) && InoutIsIn)
    {
        if (GnlCreateTimingInfo (&AuxTimingInfo))
            return (GNL_MEMORY_FULL);
        SetGnlTimingInfoHook (TimingInfo, AuxTimingInfo);
        TimingInfo = GnlTimingInfoHook (TimingInfo);
    }
}
if (GnlTimingInfoTag (TimingInfo) == Tag)
    return (GNL_OK);
SetGnlTimingInfoTag (TimingInfo, Tag);
Capa = 0.0;
```

timecomp.c

```
Fanout = 0;

if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
    (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn) )
{
    if (!BListSize (G_PileOfComponent))
    {
        /* Take the constraint from user (for later) */
        /*if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
            (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn)) */
        {
            SetGnlTimingInfoCapa (TimingInfo, Capa);
            SetGnlTimingInfoFanout (TimingInfo, 1);
            return (GNL_OK);
        }
    }
    else
    {
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                    BListSize (G_PileOfComponent)-1);

        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                              &Actual);
        if (!AuxAssocVar)
        {
            if (BListAddElt (G_PileOfComponent, InstOfGnl))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
        if (GnlComputeCapaAndFanoutFromVar (Actual, Lib, Tag, 0, 0))
            return (GNL_MEMORY_FULL);

        if (GnlTimingInfoCorrespToCurrentPathFromVar (Actual, &AuxTimingInfo,
                                                       &Key, &Rank))
            return (GNL_MEMORY_FULL);

        Capa += GnlTimingInfoCapa (AuxTimingInfo);
        Fanout += GnlTimingInfoFanout (AuxTimingInfo);

        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
    }
}

if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (LeftVarAssigneds, i);
    if (GnlComputeCapaAndFanoutFromVar (AuxVar, Lib, Tag, 0, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar, &AuxTimingInfo,
                                                   &Key,
                                                   &Rank))
```

timecomp.c

```
    return (GNL_MEMORY_FULL);

    Capa += GnlTimingInfoCapa (AuxTimingInfo);
    Fanout += GnlTimingInfoFanout (AuxTimingInfo);
}
BListQuickDelete (&LeftVarAssigneds);

for (i=0; i < BListSize (AssocDests); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocDests, i);
    if ((GnlStatus = GnlComputeCapaAndFanoutFromAssoc (AuxAssocVar, &Capa,
                                                &Fanout, Lib, Var, Tag)))
        return (GnlStatus);
}
BListQuickDelete (&AssocDests);

SetGnlTimingInfoCapa(TimingInfo, Capa);
SetGnlTimingInfoFanout(TimingInfo, Fanout);

return (GNL_OK);
}

/*-----*/
/* GnlComputeCapaAndFanoutFromGnl */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromGnl :
   - For a given GNL (Gnl) :
     - Hierarchically compute the output capacitance and the fanout on each
       GNL_VAR.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromGnl (GNL           Gnl,
                                             LIBC_LIB     Lib,
                                             int          Tag)
{
    GNL_VAR      Var;
    int          i, j, J;
    GNL_STATUS   GnlStatus;
    GNL          GnlCompoI;
    GNL_COMPONENT ComponentI;
    BLIST         Components, BucketI;

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BlistElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
/*      if (StringIdentical (GnlName (GnlCompoI), "cat_register_block-rtl"))
            J=j; */

            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlComputeCapaAndFanoutFromGnl (GnlCompoI, Lib, Tag))

```

timecomp.c

```
    return (GnlStatus);
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
}
}

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;

        if (StringIdentical (GnlVarName (Var), "F376"))
            J=j;

        if (GnlComputeCapaAndFanoutFromVar (Var, Lib, Tag, 1, 0))
            return (GNL_MEMORY_FULL);
        if (GnlVarDir (Var) == GNL_VAR_INOUT)
            if (GnlComputeCapaAndFanoutFromVar (Var, Lib, Tag, 0, 0))
                return (GNL_MEMORY_FULL);
    }
}

return (GNL_OK);
}

/*-----
 * GnlComputeCapaAndFanoutFromNetwork
 *-----
 */
/* Doc procedure GnlComputeCapaAndFanoutFromNetwork :
 - For a given Network (GNL_NETWORK) :
   - Hierarchically compute the output capacitance and the fanout on each
     GNL_VAR.

 *-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromNetwork (GNL_NETWORK Nw,
                                                LIBC_LIB      Lib)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    int          Tag;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
    Tag = GnlNetworkTag (Nw);
    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlComputeCapaAndFanoutFromGnl (TopGnl, Lib, Tag)))
        return (GnlStatus);

    BListQuickDelete (&G_PileOfComponent);
    return (GNL_OK);
}
```

timecomp.c

```
}

/*-----*/
void GnlGetArrivalTimeRiseAndFallSwitchTimingSense (LIBC_PIN PinIn, LIBC_PIN
PinOut,
                                    float DelayR, float DelayF,
                                    float TimeRiseIn, float TimeFallIn,
                                    float *TimeRise, float *TimeFall)
{
    LIBC_TIMING          Timing;

    *TimeRise = *TimeFall = 0.0;
    if (!DelayR && !DelayF)
        return;

    Timing = LibGetArcTiming (PinIn, PinOut);
    if (!Timing)
        return;

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *TimeRise = DelayR + TimeRiseIn;
            *TimeFall = DelayF + TimeFallIn;
            break;

        case NEGATIVE_UNATE_E:
            *TimeRise = DelayR + TimeFallIn;
            *TimeFall = DelayF + TimeRiseIn;
            break;

        case NON_UNATE_E:
            *TimeRise = DelayR + TimeRiseIn;
            if (TimeRiseIn < TimeFallIn)
                *TimeRise = DelayR + TimeFallIn;

            *TimeFall = DelayF + TimeFallIn;
            if (TimeFallIn < TimeRiseIn)
                *TimeFall = DelayF + TimeRiseIn;
            break;
    }

}

/*-----*/
/* GnlComputeArrivalTimeAtOutputOfCombCell */ */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeAtOutputOfCombCell :
   - For a given a GNL_USER_COMPONENT (UserCompo) generic cell and
     GNL_ASSOC (AssocOutput) (output GNL_ASSOC of UserCompo) :
       - Compute the arrival time (falling and rising)
       - Compute the transition(falling and rising)
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeAtOutputOfCombCell(GNL_USER_COMPONENT
UserCompo,
                                                GNL_ASSOC AssocOutput,
```

```

        float *MaxTimeRise,
        float *MaxTimeFall,
        float *MaxOutTransRise,
        float *MaxOutTransFall,
        LIBC_LIB     Lib,
        int         Tag,
        int         WithRec)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR       Var, Vari;
    GNL_TIMING_INFO TimingI;
    float         InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall, Time, MaxTime;
    float         Length, WireCapa, WireResistance, OutCapa;
    BLIST         Interface;
    GNL_STATUS    GnlStatus;
    unsigned int   Key;

    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
    Var = GnlAssocActualPort (AssocOutput);
    Formal = (char *) GnlAssocFormalPort (AssocOutput);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E)
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlUserComponentInterface (UserCompo);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        Vari = GnlAssocActualPort (AssocI);
        if (GnlVarIsVar (Vari))
            if (GnlVarIsVdd (Vari) || GnlVarIsVss (Vari))
                continue;

        Formal = (char *) GnlAssocFormalPort (AssocI);
        if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
            continue;
        if (LibPinDirection (PinIn) != INPUT_E)
            continue;
        if (!LibGetArcTiming (PinIn, PinOut))
            continue;
    }
}

```

timecomp.c

```
if (WithRec)
    if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (VarI) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);

    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                    Fanout, Cell, PinIn, PinOut,
                    &DelayR, &OutTransR,
                    &DelayF, &OutTransF);
    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                    DelayR, DelayF,
                                                    GnlTimingInfoArrivalRise (TimingI),
                                                    GnlTimingInfoArrivalFall (TimingI),
                                                    &TimeRise, &TimeFall);

    MaxTime = *MaxTimeRise;
    if (MaxTime < *MaxTimeFall)
        MaxTime = *MaxTimeFall;
    Time = TimeRise;
    if (Time < TimeFall)
        Time = TimeFall;

    if (MaxTime < Time)
    {
        *MaxTimeRise = TimeRise;
        *MaxTimeFall = TimeFall;
        *MaxOutTransRise = OutTransR;
        *MaxOutTransFall = OutTransF;
    }
}
return (GNL_OK);
}

/*-----
/* GnlComputeArrivalTimeFrom3State
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFrom3State :
   - For a given GNL_ASSOC (Assoc) :
      - Compute the arrival time (falling and rising)
      - Compute the transition(falling and rising)
      on the net "GnlAssocActualPort (Assoc)".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFrom3State (GNL_ASSOC Assoc,
                                             float *MaxTimeRise,
                                             float *MaxTimeFall,
                                             float *MaxOutTransRise,
                                             float *MaxOutTransFall,
                                             LIBC_LIB     Lib,
                                             int         Tag,
```

```

int           WithRec)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int            i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR        Var, Vari;
    GNL_TIMING_INFO TimingI;
    float          InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall, MaxTime, Time;
    float          Length, WireCapa, WireResistance, OutCapa;
    BLIST          Interface;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_STATUS     GnlStatus;
    unsigned int   Key;

    TriStateCompo = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlTriStateInterface (TriStateCompo);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        if (!AssocI)
            continue;
        Vari = GnlAssocActualPort (AssocI);

        if (!Vari)
            continue;

        if (GnlVarIsVar (Vari))
            if (GnlVarIsVdd (Vari) || GnlVarIsVss (Vari))
                continue;

        if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
            continue;
}

```

timecomp.c

```
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection (PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if (WithRec)
if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
    return (GnlStatus);
if (GnlVarDir (VarI) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);

InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
    Fanout, Cell, PinIn, PinOut,
    &DelayR, &OutTransR,
    &DelayF, &OutTransF);
GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
    DelayR, DelayF,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),
    &TimeRise, &TimeFall);

MaxTime = *MaxTimeRise;
if (MaxTime < *MaxTimeFall)
    MaxTime = *MaxTimeFall;
Time = TimeRise;
if (Time < TimeFall)
    Time = TimeFall;

if (MaxTime < Time)
{
    *MaxTimeRise = TimeRise;
    *MaxTimeFall = TimeFall;
    *MaxOutTransRise = OutTransR;
    *MaxOutTransFall = OutTransF;
}
}
return (GNL_OK);

}
/*-----*/
/* GnlComputeArrivalTimeFromSeqCell */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromSeqCell :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the arrival time (falling and rising)
     - Compute the transition(falling and rising)
   on the net "GnlAssocActualPort (Assoc)".
*/

```

```

/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromSeqCell (GNL_ASSOC      Assoc,
                                              float *MaxTimeRise,
                                              float *MaxTimeFall,
                                              float *MaxOutTransRise,
                                              float *MaxOutTransFall,
                                              LIBC_LIB     Lib,
                                              int         Tag,
                                              int         WithRec)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR        Var, Vari;
    GNL_TIMING_INFO TimingI;
    float          InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall;
    float          Length, WireCapa, WireResistance, OutCapa;
    BLIST          Interface;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_STATUS     GnlStatus;
    unsigned int   Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlSequentialCompoInterface (SeqCompo);
/*    for (i=0; i < BListSize (Interface); i++) */
    {
/*        AssocI = (GNL_ASSOC)BListElt (Interface, i); */
        AssocI = GnlSequentialCompoClockAssoc (SeqCompo);
        if (!AssocI)
            return (GNL_OK);
/*        continue; */
        Vari = GnlAssocActualPort (AssocI);
        if (!Vari)

```

```

        return (GNL_OK);
/*         continue; */

    if (GnlVarIsVar (VarI))
        if (GnlVarIsVdd (VarI) || GnlVarIsVss (VarI))
            return (GNL_OK);
/*         continue; */

    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
/*         continue; */
    if (LibPinDirection(PinIn) != INPUT_E)
        return (GNL_OK);
/*         continue; */
    if (!LibGetArcTiming (PinIn, PinOut))
        return (GNL_OK);
/*         continue; */
    if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                    Fanout, Cell, PinIn, PinOut,
                    &DelayR, &OutTransR,
                    &DelayF, &OutTransF);

    if (*MaxOutTransRise < OutTransR)
        *MaxOutTransRise = OutTransR ;
    if (*MaxOutTransFall < OutTransF)
        *MaxOutTransFall = OutTransF;

    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoArrivalRise (TimingI),
                                                GnlTimingInfoArrivalFall (TimingI),
                                                &TimeRise, &TimeFall);

    if (*MaxTimeRise < TimeRise)
        *MaxTimeRise = TimeRise;
    if (*MaxTimeFall < TimeFall)
        *MaxTimeFall = TimeFall;
    }

    return (GNL_OK);

}

/*
 *-----*/
/* GnlComputeArrivalTimeFromUserCompo */ 
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromUserCompo :
 - For a given GNL_ASSOC (Assoc) :
   - Compute the arrival time (falling and rising)

```

timecomp.c

```
- Compute the transition(falling and rising)
on the net "GnlAssocActualPort (Assoc)"; this net is connected to
hierarchycal block or combinational cell
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromUserCompo (GNL_ASSOC      Assoc,
                                                float        *TimeRise,
                                                float        *TimeFall,
                                                float        *TransRise,
                                                float        *TransFall,
                                                LIBC_LIB     Lib,
                                                GNL_VAR      Actual,
                                                int          Tag,
                                                int          WithRec)
{
    GNL_VAR          Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS       GnlStatus;
    int              Rank;
    unsigned int     Key;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);

    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
    {
        /* It corresponds to a LIBC library cell. */
        if (GnlUserComponentCellDef (UserCompo))
        {
            if (GnlStatus = GnlComputeArrivalTimeAtOutputOfCombCell (UserCompo,
                                                                    Assoc, TimeRise,
                                                                    TimeFall, TransRise,
                                                                    TransFall, Lib,
                                                                    Tag, WithRec))
                return (GnlStatus);

            return (GNL_OK);
        }
        else
        /* It is a black box without any definition so we stop */
        /* at its boundary. */
        {
            fprintf (stderr,
                     " WARNING: black box <%s %s> may modify final estimation\n",
                     GnlUserComponentName (UserCompo),
                     GnlUserComponentInstName (UserCompo));
        }
        return (GNL_OK);
    }

    /* If the actual net is a VDD or a VSS then no timing to compute */
    if (GnlVarIsVar (Actual))
        if (GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))
        {
            *TimeRise = 0.0;
```

timecomp.c

```
*TimeFall = 0.0;
*TransRise = 0.0;
*TransFall = 0.0;
return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
                                                &AuxFormal))
    return (GnlStatus);
if (GnlComputeArrivalTimeFromVar (AuxFormal, Lib, Tag, WithRec, 0, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal, &TimingInfo,
&Key,
    &Rank))
    return (GNL_MEMORY_FULL);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*TimeRise = GnlTimingInfoArrivalRise (TimingInfo);
*TimeFall = GnlTimingInfoArrivalFall (TimingInfo);
*TransRise = GnlTimingInfoTransitionRise (TimingInfo);
*TransFall = GnlTimingInfoTransitionFall (TimingInfo);

    return (GNL_OK);
}
/*-----*/
/* GnlComputeArrivalTimeFromAssoc */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromAssoc :
   - For a given GNL_ASSOC (Assoc) :
   - For a given GNL_ASSOC (Assoc) :
      - Compute the arrival time (falling and rising)
      - Compute the transition(falling and rising)
      on the net "GnlAssocActualPort (Assoc)".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromAssoc (GNL_ASSOC Assoc,
                                             float *TimeRise,
                                             float *TimeFall,
                                             float *TransRise,
                                             float *TransFall,
                                             LIBC_LIB Lib,
                                             GNL_VAR Actual,
                                             int Tag,
                                             int WithRec)
{
    GNL_STATUS GnlStatus;
    int Rank;
    GNL_COMPONENT GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {
```

```

    case GNL_SEQUENTIAL_COMPO:
        *TimeRise = *TimeFall = *TransRise = *TransFall = 0.0;
        if (GnlStatus = GnlComputeArrivalTimeFromSeqCell (Assoc,
TimeRise,
                                TimeFall, TransRise, TransFall, Lib, Tag, WithRec))
            return (GnlStatus);
        break;

    case GNL_USER_COMPO:
        if (GnlStatus = GnlComputeArrivalTimeFromUserCompo (Assoc,
                                TimeRise, TimeFall,
                                TransRise, TransFall, Lib,
                                Actual, Tag, WithRec))
            return (GnlStatus);
        break;

    case GNL_TRISTATE_COMPO:
        if (GnlStatus = GnlComputeArrivalTimeFrom3State (Assoc,
                                TimeRise, TimeFall,
                                TransRise, TransFall, Lib, Tag,
                                WithRec))
            return (GnlStatus);
        break;

    case GNL_MACRO_COMPO:
        break;

    default:
        GnlError (12 /* Unknown component */);
    }

    return (GNL_OK);
}
/*-----*/
/* GnlComputeSetupHoldTimeFromSeqCompo */
/*-----*/
GNL_STATUS GnlComputeSetupHoldTimeFromSeqCompo (
    GNL_SEQUENTIAL_COMPONENT SeqCompo,
    char *PinInName,
    GNL_TIMING_INFO TimingInfoOfPin,
    float *TimeRise,
    float *TimeFall,
    LIBC_LIB Lib,
    int WithRec)
{
    LIBC_CELL Cell;
    LIBC_PIN PinClock, PinIn;
    int Fanout, Rank;
    GNL_ASSOC AssocClock;
    GNL_VAR VarClock;
    char *ClockName;
    GNL_TIMING_INFO TimingClock;
    float InTransRise, InTransFall, OutTransR, OutTransF;
    float Length, WireCapa, OutCapa;
    BLIST Interface;
    GNL_STATUS GnlStatus;
}

```

timecomp.c

```
unsigned int          Key;

*TimeRise = 0.0;
*TimeFall = 0.0;
Cell = GnlSeqCompoInfoCell (SeqCompo);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, PinInName)))
    return (GNL_OK);
AssocClock = GnlSequentialCompoClockAssoc (SeqCompo);
ClockName = (char *) GnlAssocFormalPort (AssocClock);
VarClock = GnlAssocActualPort (AssocClock);
if (GnlVarIsVar (VarClock))
    if (GnlVarIsVdd (VarClock) || GnlVarIsVss (VarClock))
        return (GNL_OK);

if (!(PinClock = LibGetPinFromNameAndCell (Cell, ClockName)))
    return (GNL_OK);

if (GnlComputeArrivalTimeFromVar (VarClock, Lib, 0, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);
if (GnlTimingInfoCorrespToCurrentPathFromVar (VarClock,
                                              &TimingClock, &Key, &Rank))
    return (GNL_MEMORY_FULL);
InTransRise = GnlTimingInfoTransitionRise (TimingClock);
InTransFall = GnlTimingInfoTransitionFall (TimingClock);

Fanout = GnlTimingInfoFanout (TimingInfoOfPin);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingInfoOfPin);

LibDelayArcCellRiseSetup (InTransRise, InTransFall, OutCapa, Cell, PinIn,
                          PinClock, TimeRise);
LibDelayArcCellFallSetup (InTransRise, InTransFall, OutCapa, Cell, PinIn,
                          PinClock, TimeFall);

return (GNL_OK);
}

/*-----*/
/* GnlComputeSetupHoldTimeFromVar */
/*-----*/
GNL_STATUS GnlComputeSetupHoldTimeFromVar (GNL_VAR      Var,
                                             GNL_TIMING_INFO   TimingInfoOfVar,
                                             LIBC_LIB   Lib,
                                             float      *MaxSetupTimeR,
                                             float      *MaxSetupTimeF)
{
    GNL_ASSOC      AssocI;
    GNL_STATUS     GnlStatus;
    BLIST          AssocDests, LeftVarAssigneds;
    GNL_COMPONENT   GnlCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    int             i;
    float           MaxTimeR, MaxTimeF, TimeRise, TimeFall;
    char            *Formal;
```

timecomp.c

```
if (GnlGetDestinationsOfVar (Var, &AssocDests, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

BListQuickDelete (&LeftVarAssigneds);

*MaxSetupTimeR = *MaxSetupTimeF = 0.0;
for (i=0; i < BListSize (AssocDests); i++)
{
    AssocI = (GNL_ASSOC) BListElt (AssocDests, i);
    GnlCompo = GnlAssocTraversalInfoComponent (AssocI);
    if (GnlComponentType (GnlCompo) != GNL_SEQUENTIAL_COMPO)
        continue;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlCompo;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (GnlStatus = GnlComputeSetupHoldTimeFromSeqCompo (SeqCompo, Formal,
                TimingInfoOfVar, &TimeRise, &TimeFall, Lib, 0))
        return (GnlStatus);
    if (*MaxSetupTimeR < TimeRise)
        *MaxSetupTimeR = TimeRise;
    if (*MaxSetupTimeF < TimeFall)
        *MaxSetupTimeF = TimeFall;
}
BListQuickDelete (&AssocDests);

return (GNL_OK);
}

/*-----*/
/* GnlComputeArrivalTimeFromVar */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromVar :
   - For a given GNL_VAR (Var) :
      - Compute the arrival time (falling and rising) of
        the net "Var".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromVar (GNL_VAR Var,
                                         LIBC_LIB Lib,
                                         int Tag,
                                         int WithRec,
                                         int InoutIsIn,
                                         int VarIsInout)
{
    GNL_VAR AuxVar, Actual;
    GNL_ASSOC AuxAssocVar;
    int i, Fanout, Rank, J;
    GNL_STATUS GnlStatus;
    BLIST AssocSources, RightVarAssigneds;
    GNL_TIMING_INFO TimingInfo, AuxTimingInfo;
    GNL_USER_COMPONENT InstOfGnl;
    float TimeRise, TimeFall, TransRise, TransFall, MaxTime,
          Time;
    float MaxTimeRise, MaxTimeFall, MaxTransRise, MaxTransFall;
    float MaxSetupTimeR, MaxSetupTimeF;
    unsigned int Key;
    int AuxVarIsInout;
```

```

if (StringIdentical (GnlVarName (Var), "F376"))
    J=i;
if (GnlVarIsVar (Var))
    if (GnlVarIsVdd (Var) || GnlVarIsVss (Var))
        return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if ((GnlVarDir (Var) == GNL_VAR_INOUT) && InoutIsIn)
    TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);

if (GnlTimingInfoTag (TimingInfo) == Tag)
    return (GNL_OK);
SetGnlTimingInfoTag (TimingInfo, Tag);

TimeRise = TimeFall = MaxTimeRise = MaxTimeFall = 0.0;
TransRise = TransFall = MaxTransRise = MaxTransFall = 0.0;

if (GnlVarDir (Var) == GNL_VAR_INPUT ||
    (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn) )
{
    if (!BListSize (G_PileOfComponent))
    {
        /* Take the constraint from user (for later) */
        /*if (GnlVarDir (Var) == GNL_VAR_INPUT ||
           (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)) */
        {
            SetGnlTimingInfoArrivalRise (TimingInfo, MaxTimeRise);
            SetGnlTimingInfoArrivalFall (TimingInfo, MaxTimeFall);
            SetGnlTimingInfoTransitionRise (TimingInfo, MaxTransRise);
            SetGnlTimingInfoTransitionFall (TimingInfo, MaxTransFall);

            return (GNL_OK);
        }
    }
    else
    {
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                              &Actual);
        if (!AuxAssocVar)
        {
            if (BListAddElt (G_PileOfComponent, InstOfGnl))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
        if (!(GnlVarIsVar (Actual) &&
              (GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))))
        {
            AuxVarIsInout = (GnlVarDir (Var) == GNL_VAR_INOUT);
            if (GnlComputeArrivalTimeFromVar (Actual, Lib, Tag,

```

```

        WithRec, 1, AuxVarIsInout))
    return (GNL_MEMORY_FULL);
    if (GnlTimingInfoCorrespToCurrentPathFromVar
(AuxTimingInfo,
    &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Actual) == GNL_VAR_INOUT)
        TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook
(AuxTimingInfo);
    MaxTimeRise = GnlTimingInfoArrivalRise (AuxTimingInfo);
    MaxTimeFall = GnlTimingInfoArrivalFall (AuxTimingInfo);
    MaxTransRise = GnlTimingInfoTransitionRise (AuxTimingInfo);
    MaxTransFall = GnlTimingInfoTransitionFall (AuxTimingInfo);
}
if (BListAddElt (G_PileOfComponent, InstofGnl))
    return (GNL_MEMORY_FULL);
}
}

if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (RightVarAssigneds, i);

    if (GnlVarIsVar (AuxVar))
        if (GnlVarIsVdd (AuxVar) || GnlVarIsVss (AuxVar))
            continue;
    if (GnlVarIsVar (AuxVar))
        if (GnlVarIsVdd (AuxVar) || GnlVarIsVss (AuxVar))
            continue;

        if (GnlComputeArrivalTimeFromVar (AuxVar, Lib, Tag, WithRec, 1, 0))
            return (GNL_MEMORY_FULL);

        if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar, &AuxTimingInfo,
            &Key, &Rank))
            return (GNL_MEMORY_FULL);

        MaxTime = MaxTimeRise;
        if (MaxTime < MaxTimeFall)
            MaxTime = MaxTimeFall;
        Time = GnlTimingInfoArrivalRise (AuxTimingInfo);
        if (Time < GnlTimingInfoArrivalFall (AuxTimingInfo))
            Time = GnlTimingInfoArrivalFall (AuxTimingInfo);

        if (MaxTime < Time)
        {
            MaxTimeRise = GnlTimingInfoArrivalRise (AuxTimingInfo);
            MaxTimeFall = GnlTimingInfoArrivalFall (AuxTimingInfo);
            MaxTransRise = GnlTimingInfoTransitionRise (AuxTimingInfo);
            MaxTransFall = GnlTimingInfoTransitionFall (AuxTimingInfo);
        }
}
BListQuickDelete (&RightVarAssigneds);

```

timecomp.c

```
for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    TimeRise = TimeFall = 0.0;
    TransRise = TransFall = 0.0;
    if ((GnlStatus = GnlComputeArrivalTimeFromAssoc (AuxAssocVar, &TimeRise,
                                                       &TimeFall, &TransRise,
                                                       &TransFall, Lib, Var,
                                                       Tag, WithRec)))
        return (GnlStatus);

    MaxTime = MaxTimeRise;
    if (MaxTime < MaxTimeFall)
        MaxTime = MaxTimeFall;
    Time = TimeRise;
    if (Time < TimeFall)
        Time = TimeFall;

    if (MaxTime < Time)
    {
        MaxTimeRise = TimeRise;
        MaxTimeFall = TimeFall;
        MaxTransRise = TransRise;
        MaxTransFall = TransFall;
    }
}
BListQuickDelete (&AssocSources);

SetGnlTimingInfoArrivalRise (TimingInfo, MaxTimeRise);
SetGnlTimingInfoArrivalFall (TimingInfo, MaxTimeFall);
SetGnlTimingInfoTransitionRise (TimingInfo, MaxTransRise);
SetGnlTimingInfoTransitionFall (TimingInfo, MaxTransFall);

/* *MaxSetupTimeR = *MaxSetupTimeF = 0.0;
GnlComputeSetupHoldTimeFromVar (Var, TimingInfo, Lib,
                                 &MaxSetupTimeR, &MaxSetupTimeF);
SetGnlTimingInfoArrivalRise (TimingInfo,
                             GnlTimingInfoArrivalRise (TimingInfo) + MaxSetupTimeR);
SetGnlTimingInfoArrivalFall (TimingInfo,
                            GnlTimingInfoArrivalFall (TimingInfo) + MaxSetupTimeF);

if (MaxSetupTimeR || MaxSetupTimeF)
    if (GnlStatus = GnlUpdateArrTimeOfRightVarAssigneds (Var))
        return (GnlStatus);*/

return (GNL_OK);
}

/*
 * TimePutAssocSourceOfVarOnRight
 */
static void TimePutAssocSourceOfVarOnRight (GNL_VAR Var)
{
    int      i, j, el;
    GNL_ASSOC AssocI;
    BLIST      AssocSources, RightVarAssigneds, ListAssoc;
```

```

GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds);
ListAssoc = GnlVarTraversalInfoListAssoc (Var);
for (i=0; i < BListSize (AssocSources); i++)
{
    AssocI = (GNL_ASSOC)BListElt (AssocSources, i);
    j = BListMemberOfList (ListAssoc, AssocI, IntIdentical);

    el = BListElt (ListAssoc, BListSize (ListAssoc) -1);
    BListElt (ListAssoc, BListSize (ListAssoc) -1) = BListElt (ListAssoc, j -
1);
    BListElt (ListAssoc, j -1) = el;
}
BListQuickDelete (&AssocSources);
BListQuickDelete (&RightVarAssigneds);
}
/*-----*/
/* GnlComputeArrivalTimeFromGnl */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromGnl :
   - For a given GNL (Gnl) :
      - Hierarchically compute the arrival time (falling and rising) on each
        net GNL_VAR.
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromGnl (GNL Gnl,
                                         LIBC_LIB Lib,
                                         int Tag)
{
    GNL_VAR Var;
    int i, j, J;
    GNL_STATUS GnlStatus;
    GNL GnlCompoI;
    GNL_COMPONENT ComponentI;
    BLIST Components, BucketI;

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlComputeArrivalTimeFromGnl (GnlCompoI, Lib, Tag))
                return (GnlStatus);
            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        }
    }

    for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
    {
        BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
        for (j=0; j < BListSize (BucketI); j++)

```

timecomp.c

```
{  
    Var = (GNL_VAR)BListElt (BucketI, j);  
  
    if (!Var)  
        continue;  
  
    if (GnlVarIsVar (Var))  
        if (GnlVarIsVdd (Var) || GnlVarIsVss (Var))  
            continue;  
  
    TimePutAssocSourceOfVarOnRight (Var);  
    if (StringIdentical (GnlVarName (Var), "F376"))  
        J=i;  
    if (GnlComputeArrivalTimeFromVar (Var, Lib, Tag, 1, 0, 0))  
        return (GNL_MEMORY_FULL);  
    }  
}  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlComputeArrivalTimeFromNetwork */  
/*-----*/  
/* Doc procedure GnlComputeArrivalTimeFromNetwork :  
   - For a given Network (GNL_NETWORK) :  
     - Hierarchically compute the arrival time (falling and rising) on each  
       net GNL_VAR.  
-----*/  
GNL_STATUS GnlComputeArrivalTimeFromNetwork (GNL_NETWORK Nw,  
                                              LIBC_LIB Lib)  
{  
    GNL          TopGnl;  
    GNL_STATUS   GnlStatus;  
    int         Tag;  
  
    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);  
    Tag = GnlNetworkTag (Nw);  
    TopGnl = GnlNetworkTopGnl (Nw);  
  
    if (BListCreate (&G_PileOfComponent))  
        return (GNL_MEMORY_FULL);  
  
    if ((GnlStatus = GnlComputeArrivalTimeFromGnl (TopGnl, Lib, Tag)))  
        return (GnlStatus);  
  
    BListQuickDelete (&G_PileOfComponent);  
    return (GNL_OK);  
}  
/*-----*/  
/*-----*/
```

```

/*
GnlGetRequiredTimeRiseAndFallSwitchTimingSence (LIBC_PIN PinIn, LIBC_PIN
PinOut,
                                              float DelayR, float DelayF,
                                              float TimeRiseOut, float TimeFallOut,
                                              float *TimeRise, float *TimeFall)
{
    LIBC_TIMING Timing;

    *TimeRise = *TimeFall = MAX_FLOAT;
    if (!DelayR && !DelayF)
        return;

    Timing = LibGetArcTiming (PinIn, PinOut);

    if (!Timing)
        return;

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *TimeRise = TimeRiseOut - DelayR;
            *TimeFall = TimeFallOut - DelayF;
            break;

        case NEGATIVE_UNATE_E:
            *TimeFall = TimeRiseOut - DelayR;
            *TimeRise = TimeFallOut - DelayF;
            break;

        case NON_UNATE_E:
            *TimeRise = TimeRiseOut - DelayR;
            if (*TimeRise > TimeFallOut - DelayF)
                *TimeRise = TimeFallOut - DelayF;

            *TimeFall = TimeFallOut - DelayF;
            if (*TimeFall > TimeRiseOut - DelayR)
                *TimeFall = TimeRiseOut - DelayR;
            break;
    }

}

/*
/* GnlComputeRequiredTimeAtOutputOfCombCell */
/*
/* Doc procedure GnlComputeRequiredTimeAtOutputOfCombCell :
   - For a given a GNL_USER_COMPONENT (UserCompo) generic cell and
     GNL_ASSOC (AssocOutput) (output GNL_ASSOC of UserCompo) :
       - Compute the required time (falling and rising)
*/
/*
GNL_STATUS GnlComputeRequiredTimeAtOutputOfCombCell(

```

```

        GNL_USER_COMPONENT UserCompo,
        GNL_ASSOC AssocInput,
        float *MinTimeRise,
        float *MinTimeFall,
        LIBC_LIB    Lib,
        int       Tag,
        int       WithRec)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR       Var, Vari;
    GNL_TIMING_INFO TimingI;
    float         InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall, MinTime, Time;
    float         Length, WireCapa, WireResistance, OutCapa;
    BLIST         Interface;
    GNL_STATUS    GnlStatus;
    unsigned int   Key;

    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
    Var = GnlAssocActualPort (AssocInput);
    Formal = (char *) GnlAssocFormalPort (AssocInput);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    if (GnlVarDir (Var) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    Interface = GnlUserComponentInterface (UserCompo);
    for (i=0; i < BlistSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        Vari = GnlAssocActualPort (AssocI);
        if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
            continue;
        Formal = (char *) GnlAssocFormalPort (AssocI);
        if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
            continue;
        if (LibPinDirection(PinOut) != OUTPUT_E)
            continue;
        if (!LibGetArcTiming (PinIn, PinOut))
            continue;
        if (WithRec)
            if (GnlComputeRequiredTimeFromVar (Vari, Lib, Tag, WithRec, 0, 0))

```

timecomp.c

```
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                    Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Cell, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);

    GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                    DelayR, DelayF,
                                                    GnlTimingInfoRequiredRise (TimingI),
                                                    GnlTimingInfoRequiredFall (TimingI),
                                                    &TimeRise, &TimeFall);

    MinTime = *MinTimeRise;
    if (MinTime > *MinTimeFall)
        MinTime = *MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    if (MinTime > Time)
    {
        *MinTimeRise = TimeRise;
        *MinTimeFall = TimeFall;
    }
}
return (GNL_OK);
}

/*-----
/* GnlComputeRequiredTimeFromUserCompo
*/
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromUserCompo :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the required time (falling and rising)
       on the net "GnlAssocActualPort (Assoc)".
     - This GNL_ASSOC is connected to a user component.
*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromUserCompo (GNL_ASSOC Assoc,
                                                float    *TimeRise,
                                                float    *TimeFall,
                                                LIBC_LIB Lib,
                                                GNL_VAR Actual,
                                                int      Tag,
                                                int      WithRec)
{
```

timecomp.c

```
GNL_VAR Var, Formal, AuxFormal;
GNL_USER_COMPONENT UserCompo;
GNL_TIMING_INFO TimingInfo;
GNL_STATUS GnlStatus;
int Rank;
unsigned int Key;
int AuxVarIsInout;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Var = GnlAssocActualPort (Assoc);

Formal = GnlAssocFormalPort (Assoc);
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        if (GnlStatus = GnlComputeRequiredTimeAtOutputOfCombCell (UserCompo,
                                                               Assoc, TimeRise,
                                                               TimeFall, Lib, Tag, WithRec))
            return (GnlStatus);

        return (GNL_OK);
    }
    else
    /* It is a black box without any definition so we stop */
    /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
    return (GnlStatus);

AuxVarIsInout = (GnlVarDir (Actual) == GNL_VAR_INOUT);
if (GnlComputeRequiredTimeFromVar (AuxFormal, Lib, Tag, WithRec, 1,
AuxVarIsInout))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal,
&TimingInfo, &Key, &Rank)))
    return (GnlStatus);
if (GnlVarDir (AuxFormal) == GNL_VAR_INOUT)
    TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*TimeRise = GnlTimingInfoRequiredRise (TimingInfo);
```

timecomp.c

```
*TimeFall = GnlTimingInfoRequiredFall (TimingInfo);

    return (GNL_OK);
}
/*-----*/
/* GnlComputeRequiredTimeFromSeqCell                               */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromSeqCell :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the required time (falling and rising)
       on the net "GnlAssocActualPort (Assoc)".
   - This GNL_ASSOC is connected to a sequential component.
*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromSeqCell (GNL_ASSOC      Assoc,
                                              float        *MinTimeRise,
                                              float        *MinTimeFall,
                                              LIBC_LIB Lib,
                                              int          Tag,
                                              int          WithRec)
{
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR       Var, Vari;
    GNL_TIMING_INFO TimingI;
    float         InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall, Time, MinTime;
    float         Length, WireCapa, WireResistance, OutCapa;
    BLIST         Interface;
    GNL_STATUS    GnlStatus;
    unsigned int   Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    Interface = GnlSequentialCompoInterface (SeqCompo);
    for (i=0; i < BlistSize (Interface); i++)
```

```

{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    if (!AssocI)
        continue;
    VarI = GnlAssocActualPort (AssocI);
    if (!VarI)
        continue;
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinOut) != OUTPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if (GnlComputeRequiredTimeFromVar (VarI, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                    Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Cell, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);

    GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                    DelayR, DelayF,
                                                    GnlTimingInfoRequiredRise (TimingI),
                                                    GnlTimingInfoRequiredFall (TimingI),
                                                    &TimeRise, &TimeFall);

    MinTime = *MinTimeRise;
    if (MinTime > *MinTimeFall)
        MinTime = *MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    if (MinTime > Time)
    {
        *MinTimeRise = TimeRise;
        *MinTimeFall = TimeFall;
    }
}
return (GNL_OK);
}/*-----*/

```

timecomp.c

```

/* GnlComputeRequiredTimeFrom3State */  

/*-----*/  

/* Doc procedure GnlComputeRequiredTimeFrom3State :  

 * - For a given GNL_ASSOC (Assoc) :  

 *   - Compute the required time (falling and rising)  

 *     on the net "GnlAssocActualPort (Assoc)".  

 * - This GNL_ASSOC is connected to a tristate component.  

 */  

/*-----*/  

GNL_STATUS GnlComputeRequiredTimeFrom3State (GNL_ASSOC      Assoc,  

                                             float        *MinTimeRise,  

                                             float        *MinTimeFall,  

                                             LIBC_LIB Lib,  

                                             int         Tag,  

                                             int         WithRec)  

{  

    GNL_TRISTATE_COMPONENT      TriState;  

    LIBC_CELL       Cell;  

    LIBC_PIN        PinOut, PinIn;  

    int             i, Fanout, Rank;  

    GNL_ASSOC       AssocI;  

    char            *Formal;  

    GNL_VAR         Var, VarI;  

    GNL_TIMING_INFO TimingI;  

    float           InTransRise, InTransFall, OutTransR, OutTransF,  

                    DelayR, DelayF, TimeRise, TimeFall, MinTime, Time;  

    float           Length, WireCapa, WireResistance, OutCapa;  

    BLIST          Interface;  

    GNL_STATUS      GnlStatus;  

    unsigned int    Key;  

    TriState = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);  

    Cell = GnlTriStateCompoInfoCell (TriState);  

    Var = GnlAssocActualPort (Assoc);  

    Formal = (char *) GnlAssocFormalPort (Assoc);  

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))  

        return (GNL_OK);  

    if (LibPinDirection (PinIn) == OUTPUT_E )  

        return (GNL_OK);  

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  

                                                               &TimingI, &Key, &Rank)))  

        return (GnlStatus);  

    if (GnlVarDir (Var) == GNL_VAR_INOUT)  

        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);  

    InTransRise = GnlTimingInfoTransitionRise (TimingI);  

    InTransFall = GnlTimingInfoTransitionFall (TimingI);  

    Interface = GnlTriStateInterface (TriState);  

    for (i=0; i < BListSize (Interface); i++)  

    {  

        AssocI = (GNL_ASSOC) BListElt (Interface, i);  

        if (!AssocI)  

            continue;

```

```

VarI = GnlAssocActualPort (AssocI);
if (!VarI)
    continue;
if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
    continue;
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinOut) != OUTPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if (WithRec)
if (GnlComputeRequiredTimeFromVar (VarI, Lib, Tag, WithRec, 0, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
    return (GNL_MEMORY_FULL);

Fanout = GnlTimingInfoFanout (TimingI);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);

GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoRequiredRise (TimingI),
                                                GnlTimingInfoRequiredFall (TimingI),
                                                &TimeRise, &TimeFall);

MinTime = *MinTimeRise;
if (MinTime > *MinTimeFall)
    MinTime = *MinTimeFall;
Time = TimeRise;
if (Time > TimeFall)
    Time = TimeFall;

if (MinTime > Time)
{
    *MinTimeRise = TimeRise;
    *MinTimeFall = TimeFall;
}
}
return (GNL_OK);
}
/*-----*/
/* GnlComputeRequiredTimeFromAssoc */ */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromAssoc :

```

timecomp.c

```
- For a given GNL_ASSOC (Assoc) :
  - Compute the required time (falling and rising)
    on the net "GnlAssocActualPort (Assoc)".

*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromAssoc (GNL_ASSOC Assoc,
                                             float      *TimeRise,
                                             float      *TimeFall,
                                             LIBC_LIB Lib,
                                             GNL_VAR   Actual,
                                             int       Tag,
                                             int       WithRec)
{
    GNL_STATUS          GnlStatus;
    GNL_COMPONENT       GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            *TimeRise = *TimeFall = 0.0;
            break;
        if (GnlStatus = GnlComputeRequiredTimeFromSeqCell (Assoc,
TimeRise,
                                         TimeFall, Lib, Tag, WithRec))
            return (GnlStatus);
        break;

        case GNL_USER_COMPO:
            if (GnlStatus = GnlComputeRequiredTimeFromUserCompo (Assoc,
TimeRise, TimeFall, Lib,
                                         Actual, Tag, WithRec))
                return (GnlStatus);
            break;

        case GNL_TRISTATE_COMPO:
            if (GnlStatus = GnlComputeRequiredTimeFrom3State (Assoc,
TimeRise, TimeFall, Lib,
                                         Tag, WithRec))
                return (GnlStatus);
            break;

        case GNL_MACRO_COMPO:
            break;

        default:
            GnlError (12 /* Unknown component */);
            break;
    }

    return (GNL_OK);
}
/*-----*/
/* TimeSortListByRequiredTime                               */
/*-----*/
static void TimeSortListByRequiredTime (BLIST List, BLIST ListRequireTime)
```

timecomp.c

```
{  
    int          i, k, IndexMin, el;  
    float        Min;  
  
    k = 0;  
    while (k < BListSize (ListRequireTime))  
    {  
        Min = MAX_FLOAT;  
        IndexMin = k;  
        for (i = k; i < BListSize (ListRequireTime); i++)  
        {  
            if (*((float *) BListElt (ListRequireTime, i)) < Min)  
            {  
                IndexMin = i;  
                Min = *((float *) BListElt (ListRequireTime, i));  
            }  
        }  
        el = BListElt (ListRequireTime, IndexMin);  
        BListElt (ListRequireTime, IndexMin) = BListElt (ListRequireTime, k);  
        BListElt (ListRequireTime, k) = el;  
  
        el = BListElt (List, IndexMin);  
        BListElt (List, IndexMin) = BListElt (List, k);  
        BListElt (List, k) = el;  
        k++;  
    }  
}  
  
/*-----*/  
/* GnlComputeRequiredTimeFromVar */  
/*-----*/  
/* Doc procedure GnlComputeRequiredTimeFromVar :  
   - For a given GNL_VAR (Var) :  
     - Compute the required time (falling and rising) of  
       the net "Var".  
 */  
/*-----*/  
GNL_STATUS GnlComputeRequiredTimeFromVar (GNL_VAR      Var,  
                                         LIBC_LIB  Lib,  
                                         int       Tag,  
                                         int       WithRec,  
                                         int       InoutIsIn,  
                                         int       VarIsInout)  
{  
    GNL_VAR          AuxVar, Actual;  
    GNL_ASSOC        AuxAssocVar;  
    int              i, Fanout, Rank, Elt, J;  
    GNL_STATUS       GnlStatus;  
    BLIST           AssocDests, LeftVarAssigneds,  
ListAssoc, ListRequireTime;  
    GNL_TIMING_INFO TimingInfo, AuxTimingInfo;  
    GNL_USER_COMPONENT InstOfGnl;  
    float            TimeRise, TimeFall, Time, MinTime, *Real;  
    float            MinTimeRise, MinTimeFall;  
    float            MaxSetupTimeR, MaxSetupTimeF;  
    unsigned int     Key;
```

timecomp.c

```
int AuxVarIsInout;

if (StringIdentical (GnlVarName (Var), "F376"))
    J=i;

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if ((GnlVarDir (Var) == GNL_VAR_INOUT) && InoutIsIn)
    TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);
if (GnlTimingInfoTag (TimingInfo) == Tag)
    return (GNL_OK);
SetGnlTimingInfoTag (TimingInfo, Tag);
TimeRise = TimeFall = MinTimeRise = MinTimeFall = 0.0;

if ( (GnlVarDir (Var) == GNL_VAR_OUTPUT) ||
     (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn) )
{
    if (!BListSize (G_PileOfComponent))
    {
        /* Take the constraint from user (for later) */
        /*if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
           (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn))*/
        {
            SetGnlTimingInfoRequiredRise (TimingInfo, MinTimeRise);
            SetGnlTimingInfoRequiredFall (TimingInfo, MinTimeFall);
            return (GNL_OK);
        }
    }
    else
    {
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                          &Actual);
        if (!AuxAssocVar)
        {
            if (BListAddElt (G_PileOfComponent, InstOfGnl))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
        if (GnlComputeRequiredTimeFromVar (Actual, Lib, Tag, WithRec, 0, 0))
            return (GNL_MEMORY_FULL);
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Actual,
                        &AuxTimingInfo, &Key, &Rank)))
            return (GnlStatus);

        TimeRise = GnlTimingInfoRequiredRise (AuxTimingInfo);
        TimeFall = GnlTimingInfoRequiredFall (AuxTimingInfo);
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);

        MinTime = MinTimeRise;
        if (MinTime > MinTimeFall)
```

```

    MinTime = MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    if (MinTime > Time)
    {
        MinTimeRise = TimeRise;
        MinTimeFall = TimeFall;
    }
}

if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

ListAssoc = GnlVarTraversalInfoListLeftAssign (Var);
if (BListCreateWithSize (BListSize (LeftVarAssigneds), &ListRequireTime))
    return (GNL_MEMORY_FULL);
for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (LeftVarAssigneds, i);
    if (GnlComputeRequiredTimeFromVar (AuxVar, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar,
                                                               &AuxTimingInfo, &Key, &Rank)))
        return (GnlStatus);

    MinTime = MinTimeRise;
    if (MinTime > MinTimeFall)
        MinTime = MinTimeFall;
    Time = GnlTimingInfoRequiredRise (AuxTimingInfo);
    if (Time > GnlTimingInfoRequiredFall (AuxTimingInfo))
        Time = GnlTimingInfoRequiredFall (AuxTimingInfo);

    Real = (float *) calloc (1, sizeof (float));
    (*Real) = Time;
    if (BListAddElt (ListRequireTime, (int) Real))
        return (GNL_MEMORY_FULL);

    if (MinTime > Time)
    {
        MinTimeRise = GnlTimingInfoRequiredRise (AuxTimingInfo);
        MinTimeFall = GnlTimingInfoRequiredFall (AuxTimingInfo);
    }
}
/* Tri of ListAssoc by required time */
TimeSortListByRequiredTime (ListAssoc, ListRequireTime);
BListDelete (&ListRequireTime, BListElemFree);
BListQuickDelete (&LeftVarAssigneds);
ListAssoc = GnlVarTraversalInfoListAssoc (Var);
if (BListCreateWithSize (BListSize (AssocDest), &ListRequireTime))
    return (GNL_MEMORY_FULL);
for (i=0; i < BListSize (AssocDest); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocDest, i);
}

```

timecomp.c

```
TimeRise = TimeFall = 0.0;
if (GnlComputeRequiredTimeFromAssoc (AuxAssocVar, &TimeRise,
                                     &TimeFall, Lib, Var, Tag, WithRec))
    return (GNL_MEMORY_FULL);

MinTime = MinTimeRise;
if (MinTime > MinTimeFall)
    MinTime = MinTimeFall;
Time = TimeRise;
if (Time > TimeFall)
    Time = TimeFall;

Real = (float *) calloc (1, sizeof (float));
(*Real) = Time;
if (BListAddElt (ListRequireTime, (int) Real))
    return (GNL_MEMORY_FULL);

if (MinTime > Time)
{
    MinTimeRise = TimeRise;
    MinTimeFall = TimeFall;
}
/* Tri of ListAssoc by required time */
TimeSortListByRequiredTime (ListAssoc, ListRequireTime);
BListDelete (&ListRequireTime, BListElemFree);
BListQuickDelete (&AssocDest);

SetGnlTimingInfoRequiredRise (TimingInfo, MinTimeRise);
SetGnlTimingInfoRequiredFall (TimingInfo, MinTimeFall);

/* *MaxSetupTimeR = *MaxSetupTimeF = 0.0;
GnlComputeSetupHoldTimeFromVar (Var, TimingInfo, Lib,
                                 &MaxSetupTimeR, &MaxSetupTimeF);
SetGnlTimingInfoRequiredRise (TimingInfo,
    GnlTimingInfoRequiredRise (TimingInfo) - MaxSetupTimeR);
SetGnlTimingInfoRequiredFall (TimingInfo,
    GnlTimingInfoRequiredFall (TimingInfo) - MaxSetupTimeF);
*/
return (GNL_OK);
}

/*
 *-----*
 * GnlComputeRequiredTimeFromGnl                               */
 *-----*/
/* Doc procedure GnlComputeRequiredTimeFromGnl :
   - For a given GNL (Gnl) :
     - Hierarchically compute the required time (falling and rising) on each
       net GNL_VAR.
*/
/*
 *-----*/
GNL_STATUS GnlComputeRequiredTimeFromGnl (GNL           Gnl,
                                           LIBC_LIB      Lib,
                                           int           Tag)
{
    GNL_VAR          Var;
```

timecomp.c

```
int          i, j;
GNL_STATUS   GnlStatus;
GNL          GnlCompoI;
GNL_COMPONENT ComponentI;
BLIST        Components, BucketI;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT) BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
    GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
    if (GnlCompoI)
    {
        if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
            return (GNL_MEMORY_FULL);
        if (GnlStatus = GnlComputeRequiredTimeFromGnl (GnlCompoI, Lib, Tag))
            return (GnlStatus);
        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    }
}

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;
        if (GnlComputeRequiredTimeFromVar (Var, Lib, Tag, 1, 1, 0))
            return (GNL_MEMORY_FULL);
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlComputeRequiredTimeFromNetwork                                */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromNetwork :
   - For a given Network (GNL_NETWORK) :
     - Hierarchically compute the required time (falling and rising) on each
       net GNL_VAR. */

/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromNetwork (GNL_NETWORK Nw,
                                              LIBC_LIB   Lib)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    int          Tag;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
    Tag = GnlNetworkTag (Nw);
```

```

TopGnl = GnlNetworkTopGnl (Nw);

if (BListCreate (&G_PileOfComponent))
    return (GNL_MEMORY_FULL);

if ((GnlStatus =GnlComputeRequiredTimeFromGnl (TopGnl, Lib, Tag)))
    return (GnlStatus);

BListQuickDelete (&G_PileOfComponent);
return (GNL_OK);
}

-----*/
GnlGetSensFromArcTimingAndInputArrivalTime (LIBC_PIN PinIn, LIBC_PIN PinOut,
                                             float TimeRiseIn, float TimeFallIn,
                                             int *SensRise, int *SensFall)

{
    LIBC_TIMING          Timing;

    Timing = LibGetArcTiming (PinIn, PinOut);

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *SensRise = RISE;
            *SensFall = FALL;
            break;

        case NEGATIVE_UNATE_E:
            *SensRise = FALL;
            *SensFall = RISE;
            break;

        case NON_UNATE_E:
            *SensRise = RISE;
            if (TimeRiseIn < TimeFallIn)
                *SensRise = FALL;

            *SensFall = FALL;
            if (TimeFallIn < TimeRiseIn)
                *SensFall = RISE;
            break;
    }
}
/*-----*/
extern GNL_STATUS GnlGetCritPathFromVar (GNL_VAR      Var,
                                         LIBC_LIB      Lib,
                                         GNL_CRITICAL_PATH *CritPath,
                                         GNL_CRITICAL_PATH CritSuccessorPath,
                                         int   KindOfTime,
                                         int *IsCombPath, int *IsSeqPath, int Start,
                                         int VarIsInout);
/*
-----*/

```

timecomp.c

```
/* GnlGetCritPathAtOutputOfCombCell */  
/*-----*/  
GNL_STATUS GnlGetCritPathAtOutputOfCombCell (GNL_USER_COMPONENT UserCompo,  
                                             GNL_ASSOC    Assoc,  
                                             LIBC_LIB     Lib,  
                                             GNL_CRITICAL_PATH *CritPath,  
                                             GNL_CRITICAL_PATH CritSuccessorPath,  
                                             int *IsCombPath, int *IsSeqPath)  
{  
    LIBC_CELL      Cell;  
    LIBC_PIN       PinOut, PinIn;  
    int           i, Fanout, Rank, SensRise, SensFall;  
    GNL_ASSOC     AssocI;  
    char          *Formal;  
    GNL_VAR        Var, VarI;  
    GNL_TIMING_INFO TimingI;  
    float         InTransRise, InTransFall, OutTransR, OutTransF,  
                  DelayR, DelayF, TimeRise, TimeFall;  
    float         Length, WireCapa, WireResistance, OutCapa;  
    float         SlackRise, SlackFall, MinSlack;  
    BLIST         Interface;  
    GNL_STATUS    GnlStatus;  
    int           KindOfTime;  
    GNL_CRITICAL_PATH CritPredecessorPath;  
    unsigned int   Key;  
  
    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);  
    Var = GnlAssocActualPort (Assoc);  
    Formal = (char *) GnlAssocFormalPort (Assoc);  
  
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))  
        return (GNL_OK);  
    if (LibPinDirection (PinOut) == INPUT_E )  
        return (GNL_OK);  
  
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
                                                               &TimingI, &Key, &Rank))  
        return (GnlStatus);  
  
    Fanout = GnlTimingInfoFanout (TimingI);  
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);  
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);  
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);  
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);  
  
    Interface = GnlUserComponentInterface (UserCompo);  
    MinSlack = GnlCritPathReqTime (CritSuccessorPath) -  
               GnlCritPathArrTime (CritSuccessorPath);  
for (i=0; i < BlistSize (Interface); i++)  
{  
    AssocI = (GNL_ASSOC)BListElt (Interface, i);  
    VarI = GnlAssocActualPort (AssocI);  
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))  
        continue;  
    Formal = (char *) GnlAssocFormalPort (AssocI);  
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
```

```

        continue;
    if (LibPinDirection(PinIn) != INPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);
    if (GnlVarDir (VarI) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Cell, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);

    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                   DelayR, DelayF,
                                                   GnlTimingInfoArrivalRise (TimingI),
                                                   GnlTimingInfoArrivalFall (TimingI),
                                                   &TimeRise, &TimeFall);

    if (!GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeRise,
4) && !GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeFall,
4))
        continue;

    if ((GnlStatus = GnlCreateCritPath (CritPath)))
        return (GnlStatus);
    CritPredecessorPath = NULL;
    SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
    SetGnlCritPathFanout (*CritPath, Fanout);
    SetGnlCritPathArrTime (*CritPath,
                           GnlCritPathArrTime (CritSuccessorPath));
    SetGnlCritPathReqTime (*CritPath,
                           GnlCritPathReqTime (CritSuccessorPath));
    SetGnlCritPathKindOfTime (*CritPath,
                           GnlCritPathKindOfTime (CritSuccessorPath));
    GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
    SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                         GnlUserComponentInstName (UserCompo));
    GnlGetSensFromArcTimingAndInputArrivalTime (PinIn, PinOut,
                                                GnlTimingInfoArrivalRise (TimingI),
                                                GnlTimingInfoArrivalFall (TimingI),
                                                &SensRise,&SensFall);
    if (GnlCritPathKindOfTime (CritSuccessorPath) == RISE)
    {
        SetGnlCritPathIncr (*CritPath, DelayR);
        KindOfTime = SensRise;
    }
    else
    {
        SetGnlCritPathIncr (*CritPath, DelayF);
        KindOfTime = SensFall;
    }
}

```

```

    }

    if (GnlGetCritPathFromVar (VarI, Lib,&CritPredecessorPath,
                               *CritPath, KindOfTime,
                               IsCombPath, IsSeqPath, 1, 0))
        return (GNL_MEMORY_FULL);
    GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
}
return (GNL_OK);
}

/*-----*/
/* GnlGetCritPathFromUserCompo */
/*-----*/
GNL_STATUS GnlGetCritPathFromUserCompo (GNL_ASSOC      Assoc,
                                         LIBC_LIB      Lib,
                                         GNL_CRITICAL_PATH *CritPath,
                                         GNL_CRITICAL_PATH CritSuccessorPath,
                                         int           KindOfTime,
                                         int *IsCombPath, int *IsSeqPath)
{
    GNL_VAR          Var, Formal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS       GnlStatus;
    int             Rank;
    GNL_CRITICAL_PATH CritPredecessorPath;
    float           SlackRise, SlackFall, MinSlack;
    unsigned int     Key;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);

    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
    {
        /* It corresponds to a LIBC library cell. */
        if (GnlUserComponentCellDef (UserCompo))
        {
            if (GnlStatus = GnlGetCritPathAtOutputOfCombCell (UserCompo,
                                                               Assoc, Lib, CritPath,
                                                               CritSuccessorPath,
                                                               IsCombPath, IsSeqPath))
                return (GnlStatus);

            return (GNL_OK);
        }
        else
        /* It is a black box without any definition so we stop */
        /* at its boundary. */
        {
            fprintf (stderr,
                     " WARNING: black box <%s %s> may modify final estimation\n",
                     GnlUserComponentName (UserCompo),
                     GnlUserComponentInstName (UserCompo));
        }
    }
    return (GNL_OK);
}

```

```

}

if (BListAddElT (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Formal,
                &TimingInfo, &Key, &Rank)))
    return (Gnlstatus);

SlackRise = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
SlackFall = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
            GnlCritPathArrTime (CritSuccessorPath);
if ( !GnlFloatEqual (MinSlack, SlackRise, 4)
    && !GnlFloatEqual (MinSlack, SlackFall, 4) )
{
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    return (GNL_OK);
}

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathArrTime (*CritPath, GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath, GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath, KindOfTime);
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathIncr (*CritPath, 0.0);

if (GnlGetCritPathFromVar (Formal, Lib, &CritPredecessorPath,
                           *CritPath, KindOfTime, IsCombPath, IsSeqPath, 0, 0))
    return (GNL_MEMORY_FULL);
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

return (GNL_OK);
}

/*
 *-----*
 * GnlGetCritPathFromSeqCell
 *-----*/
GNL_STATUS GnlGetCritPathFromSeqCell (GNL_ASSOC Assoc,
                                      LIBC_LIB     Lib,
                                      GNL_CRITICAL_PATH *CritPath,
                                      GNL_CRITICAL_PATH CritSuccessorPath)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut;
    int           i, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR        Var;
    GNL_TIMING_INFO TimingI;
    float         SlackRise, SlackFall, MinSlack;
}

```

timecomp.c

```

GNL_STATUS          GnlStatus;
int                KindOfTime;
GNL_CRITICAL_PATH CritPredecessorPath;
GNL_SEQUENTIAL_COMPONENT SeqCompo;
unsigned int        Key;

SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
Cell = GnlSeqCompoInfoCell (SeqCompo);
Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinOut) != OUTPUT_E )
    return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);

MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
            GnlCritPathArrTime (CritSuccessorPath);
SlackRise = GnlTimingInfoRequiredRise (TimingI) -
            GnlTimingInfoArrivalRise (TimingI);
SlackFall = GnlTimingInfoRequiredFall (TimingI) -
            GnlTimingInfoArrivalFall (TimingI);

if (!GnlFloatEqual (MinSlack, SlackRise, 4)
    && !GnlFloatEqual (MinSlack, SlackFall, 4) )
    return (GNL_OK);

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathArrTime (*CritPath,
                      GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath,
                      GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath,
                         GnlCritPathKindOfTime (CritSuccessorPath));
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                      GnlSequentialCompoInstName (SeqCompo));
SetGnlCritPathIncr (*CritPath, GnlCritPathArrTime (*CritPath));

return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFromInputOfSeqCell */
/*-----*/
GNL_STATUS GnlGetCritPathFromInputOfSeqCell (GNL_ASSOC      Assoc,
                                             LIBC_LIB     Lib,
                                             GNL_CRITICAL_PATH *CritPath,
                                             GNL_CRITICAL_PATH CritSuccessorPath,
                                             int           KindOfTime)

```

timecomp.c

```
{  
    int             i, Rank;  
    char            *Formal;  
    GNL_VAR        Var;  
    GNL_TIMING_INFO TimingInfo;  
    float           TimeRise, TimeFall, MinSlack;  
    BLIST           Interface;  
    GNL_STATUS      GnlStatus;  
    GNL_CRITICAL_PATH CritPredecessorPath;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    int             IsCombPath, IsSeqPath;  
    unsigned int     Key;  
  
    Var = GnlAssocActualPort (Assoc);  
    Formal = (char *) GnlAssocFormalPort (Assoc);  
    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent  
(Assoc);  
  
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
                                              &TimingInfo, &Key, &Rank)))  
        return (GnlStatus);  
  
    if ((GnlStatus = GnlCreateCritPath (CritPath)))  
        return (GnlStatus);  
    CritPredecessorPath = NULL;  
    SetGnlCritPathCriticalAssoc (*CritPath, Assoc);  
    SetGnlCritPathFanout (*CritPath, GnlTimingInfoFanout (TimingInfo));  
    SetGnlCritPathKindOfTime (*CritPath, KindOfTime);  
    GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);  
    SetGnlCritPathInstNameFromUserCompo (*CritPath,  
                                         GnlSequentialCompoInstName (SeqCompo));  
  
    if (GnlStatus = GnlComputeSetupHoldTimeFromSeqCompo (SeqCompo, Formal,  
                                              TimingInfo, &TimeRise, &TimeFall, Lib, 0))  
        return (GnlStatus);  
    if (GnlGetCritPathFromVar (Var, Lib, &CritPredecessorPath,  
                           *CritPath, KindOfTime, &IsCombPath, &IsSeqPath, 1, 0))  
        return (GNL_MEMORY_FULL);  
  
    if (KindOfTime == RISE)  
    {  
        SetGnlCritPathArrTime (*CritPath, GnlTimingInfoArrivalRise  
(TimingInfo));  
        SetGnlCritPathReqTime (*CritPath, GnlTimingInfoRequiredRise  
(TimingInfo));  
        SetGnlCritPathIncr (*CritPath, TimeRise);  
    }  
    else  
    {  
        SetGnlCritPathArrTime (*CritPath, GnlTimingInfoArrivalFall  
(TimingInfo));  
        SetGnlCritPathReqTime (*CritPath, GnlTimingInfoRequiredFall  
(TimingInfo));  
        SetGnlCritPathIncr (*CritPath, TimeFall);  
    }  
}
```

timecomp.c

```
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
    return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFrom3State */
/*-----*/
GNL_STATUS GnlGetCritPathFrom3State (GNL_ASSOC Assoc,
                                      LIBC_LIB Lib,
                                      GNL_CRITICAL_PATH *CritPath,
                                      GNL_CRITICAL_PATH CritSuccessorPath,
                                      int *IsCombPath, int *IsSeqPath)
{
    LIBC_CELL Cell;
    LIBC_PIN PinOut, PinIn;
    int i, Fanout, Rank, SensRise, SensFall;
    GNL_ASSOC AssocI;
    char *Formal;
    GNL_VAR Var, VarI;
    GNL_TIMING_INFO TimingI;
    float InTransRise, InTransFall, OutTransR, OutTransF,
          DelayR, DelayF, TimeRise, TimeFall;
    float Length, WireCapa, WireResistance, OutCapa;
    float SlackRise, SlackFall, MinSlack;
    BLIST Interface;
    GNL_STATUS GnlStatus;
    int KindOfTime;
    GNL_CRITICAL_PATH CritPredecessorPath;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    unsigned int Key;

    TriStateCompo = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlTriStateInterface (TriStateCompo);
    MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
               GnlCritPathArrTime (CritSuccessorPath);
    for (i=0; i < BListSize (Interface); i++)
    {
```

```

AssocI = (GNL_ASSOC)BListElt (Interface, i);
if (!AssocI)
    continue;
VarI = GnlAssocActualPort (AssocI);
if (!VarI)
    continue;
if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
    continue;
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
    &TimingI, &Key, &Rank)))
    return (GnlStatus);
if (GnlVarDir (VarI) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
    Fanout, Cell, PinIn, PinOut,
    &DelayR, &OutTransR,
    &DelayF, &OutTransF);

GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
    DelayR, DelayF,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),
    &TimeRise, &TimeFall);

if (!GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeRise,
4)
    && !GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeFall,
4))
    continue;

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathFanout (*CritPath, Fanout);
SetGnlCritPathArrTime (*CritPath,
    GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath,
    GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath,
    GnlCritPathKindOfTime (CritSuccessorPath));
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
    GnlTriStateInstName (TriStateCompo));
GnlGetSensFromArcTimingAndInputArrivalTime (PinIn, PinOut,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),

```

```

        &SensRise, &SensFall);
    if (GnlCritPathKindOfTime (CritSuccessorPath) == RISE)
    {
        SetGnlCritPathIncr (*CritPath, DelayR);
        KindOfTime = SensRise;
    }
    else
    {
        SetGnlCritPathIncr (*CritPath, DelayF);
        KindOfTime = SensFall;
    }
    if (GnlStatus = GnlGetCritPathFromVar (VarI, Lib,
                                           &CritPredecessorPath,
                                           *CritPath, KindOfTime,
                                           IsCombPath, IsSeqPath, 1, 0))
        return (GnlStatus);
    GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
}
return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFromAssoc */
/*-----*/
GNL_STATUS GnlGetCritPathFromAssoc (GNL_ASSOC Assoc,
                                     LIBC_LIB Lib,
                                     GNL_CRITICAL_PATH *CritPath,
                                     GNL_CRITICAL_PATH CritSuccessorPath,
                                     int KindOfTime,
                                     int *IsCombPath, int *IsSeqPath)
{
    GNL_STATUS GnlStatus;
    int Rank;
    GNL_COMPONENT GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            if (GnlStatus = GnlGetCritPathFromSeqCell (Assoc, Lib,
                                                       CritPath, CritSuccessorPath))
                return (GnlStatus);
            *IsSeqPath = 1;
            break;

        case GNL_USER_COMPO:
            if (GnlStatus = GnlGetCritPathFromUserCompo (Assoc, Lib,
                                                       CritPath, CritSuccessorPath,
                                                       KindOfTime,
                                                       IsCombPath, IsSeqPath))
                return (GnlStatus);
            break;

        case GNL_TRISTATE_COMPO:
            if (GnlStatus = GnlGetCritPathFrom3State (Assoc, Lib,
                                                       CritPath, CritSuccessorPath,
                                                       IsCombPath, IsSeqPath))

```

timecomp.c

```
        return (GnlStatus);
    break;

    case GNL_MACRO_COMPO:
        break;

    default:
        GnlError (12 /* Unknown component */);
        break;
    }

    return (GNL_OK);
}
/*-----
/* GnlGetCritPathFromVar
*/
-----*/
GNL_STATUS GnlGetCritPathFromVar (GNL_VAR Var,
                                  LIBC_LIB Lib,
                                  GNL_CRITICAL_PATH *CritPath,
                                  GNL_CRITICAL_PATH CritSuccessorPath,
                                  int KindOfTime,
                                  int *IsCombPath, int *IsSeqPath, int Start,
                                  int VarIsInout)
{
    GNL_VAR          VarI, Actual;
    GNL_ASSOC        AuxAssocVar;
    int              i, Rank;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO TimingInfo, TimingI;
    float             ArrivalTime, RequiredTime,
                      SlackRise, SlackFall, MinSlack;
    GNL_USER_COMPONENT InstOfGnl;
    GNL_CRITICAL_PATH CritPredecessorPath, CritPredecessorPath2;
    BLIST            AssocSources, RightVarAssigneds;
    unsigned int      Key;
    int               AuxVarIsInout;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    if (GnlVarDir (Var) == GNL_VAR_INOUT && Start)
        TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);

    if ((GnlStatus = GnlCreateCritPath (CritPath)))
        return (GnlStatus);
    CritPredecessorPath = NULL;
    SetGnlCritPathCriticalVar (*CritPath, Var);
    if (KindOfTime == RISE)
    {
        ArrivalTime = GnlTimingInfoArrivalRise (TimingInfo);
        RequiredTime = GnlTimingInfoRequiredRise (TimingInfo);
    }
    else
    {
        ArrivalTime = GnlTimingInfoArrivalFall (TimingInfo);
        RequiredTime = GnlTimingInfoRequiredFall (TimingInfo);
    }
}
```

```

    }

    MinSlack = RequiredTime - ArrivalTime;
    SetGnlCritPathArrTime (*CritPath, ArrivalTime);
    SetGnlCritPathReqTime (*CritPath, RequiredTime);
    SetGnlCritPathKindOfTime (*CritPath, KindOfTime);
    GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
    SetGnlCritPathIncr (*CritPath, 0.0);

    if (GnlVarDir (Var) == GNL_VAR_INPUT ||
        (GnlVarDir (Var) == GNL_VAR_INOUT && Start))
    {
        if (!BListSize (G_PileOfComponent))
        {
            /*if (GnlVarDir (Var) == GNL_VAR_INPUT ||
               (GnlVarDir (Var) == GNL_VAR_INOUT && Start))*/
            {
                *IsCombPath = 1;
                return (GNL_OK);
            }
        }
        else
        {
            InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                              BListSize (G_PileOfComponent)-1);

            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock(Var,
InstOfGnl,&Actual);
            if (!AuxAssocVar)
            {
                if (BListAddElt (G_PileOfComponent, InstOfGnl))
                    return (GNL_MEMORY_FULL);
                return (GNL_OK);
            }

            if (GnlTimingInfoCorrespToCurrentPathFromVar(Actual,
&TimingI,&Key,&Rank))
                return (GNL_MEMORY_FULL);
            if (GnlVarDir (Actual) == GNL_VAR_INOUT && Start)
                TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
            SlackRise = GnlTimingInfoRequiredRise (TimingI) -
                        GnlTimingInfoArrivalRise (TimingI);
            SlackFall = GnlTimingInfoRequiredFall (TimingI) -
                        GnlTimingInfoArrivalFall (TimingI);
            MinSlack = GnlCritPathReqTime (*CritPath) -
                        GnlCritPathArrTime (*CritPath);
            if (!GnlFloatEqual (MinSlack, SlackRise, 4)
                && !GnlFloatEqual (MinSlack, SlackFall, 4) )
                return (GNL_OK);
            if ((GnlStatus = GnlCreateCritPath (&CritPredecessorPath)))
                return (GnlStatus);
            CritPredecessorPath2 = NULL;
            SetGnlCritPathCriticalAssoc (CritPredecessorPath, AuxAssocVar);
            SetGnlCritPathArrTime
(CritPredecessorPath,GnlCritPathArrTime(*CritPath));
            SetGnlCritPathReqTime
(CritPredecessorPath,GnlCritPathReqTime(*CritPath));

```

timecomp.c

```
SetGnlCritPathKindOfTime (CritPredecessorPath, KindOfTime);
GnlAddSuccessorToCritPath (CritPredecessorPath, *CritPath);
SetGnlCritPathIncr (CritPredecessorPath, 0.0);

AuxVarIsInout = (GnlVarDir (Var) == GNL_VAR_INOUT );
if (GnlGetCritPathFromVar (Actual, Lib, &CritPredecessorPath2,
                           CritPredecessorPath, KindOfTime,
                           IsCombPath, IsSeqPath, 1, AuxVarIsInout))
    return (GnlStatus);
GnlAddPredecessorToCritPath (CritPredecessorPath,
                             CritPredecessorPath2);

GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
if (BListAddElt (G_PileOfComponent, InstOfGnl))
    return (GNL_MEMORY_FULL);
SetGnlCritPathIncr (CritPredecessorPath, 0.0);
return (GNL_OK);
}
}

/* in this section we compute the Successor critical Path from Var */

if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    VarI = (GNL_VAR) BListElt (RightVarAssigneds, i);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    SlackRise = GnlTimingInfoRequiredRise (TimingInfo) -
                GnlTimingInfoArrivalRise (TimingInfo);
    SlackFall = GnlTimingInfoRequiredFall (TimingInfo) -
                GnlTimingInfoArrivalFall (TimingInfo);
    if ( GnlFloatEqual (MinSlack, SlackRise, 4) ||
        GnlFloatEqual (MinSlack, SlackFall, 4) )
    {
        if (GnlGetCritPathFromVar (VarI, Lib, &CritPredecessorPath,
                                   *CritPath, KindOfTime,
                                   IsCombPath, IsSeqPath, 1, 0))
            return (GNL_MEMORY_FULL);
        GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
    }
}

BListQuickDelete (&RightVarAssigneds);

for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    if (GnlGetCritPathFromAssoc (AuxAssocVar, Lib, &CritPredecessorPath,
                               *CritPath, KindOfTime,
                               IsCombPath, IsSeqPath))
        return (GNL_MEMORY_FULL);
```

timecomp.c

```
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);  
}  
  
BListQuickDelete (&AssocSources);  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* GnlGetMinSlackFromInputsOfSequentialInst */  
/*-----*/  
/* Doc procedure GnlGetMinSlackFromInputsOfSequentialInst :  
 - Getting the slack minimal attached to the inputs of sequential instances  
 of a given netlist (GNL).  
  
-----*/  
GNL_STATUS GnlGetMinSlackFromInputsOfSequentialInst (GNL Gnl, float  
*MinSlack)  
{  
    GNL_VAR           Var;  
    int               i, Rank;  
    GNL_STATUS        GnlStatus;  
    GNL_TIMING_INFO   TimingInfo;  
    float             Slack;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    GNL_COMPONENT     ComponentI;  
    GNL               GnlCompoI;  
    unsigned int      Key;  
  
    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)  
    {  
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);  
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)  
        {  
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);  
            if (GnlCompoI)  
            {  
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))  
                    return (GNL_MEMORY_FULL);  
                if (GnlStatus = GnlGetMinSlackFromInputsOfSequentialInst (GnlCompoI,  
                                                               MinSlack))  
                    return (GnlStatus);  
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));  
            }  
        }  
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)  
            continue;  
  
        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;  
        Var = GnlSequentialCompoInput (SeqCompo);  
        if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))  
            continue;  
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
                                                               &TimingInfo, &Key, &Rank)))  
            return (GnlStatus);  
    }
```

timecomp.c

```
/*      fprintf (stderr,"%s\t%.2f\t%.2f\t%.2f", GnlSequentialCompoInstName
(SeqCompo), GnlTimingInfoArrivalRise (TimingInfo), GnlTimingInfoRequiredRise
(TimingInfo), GnlTimingInfoCapa (TimingInfo));
*/
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
                    GnlTimingInfoArrivalRise (TimingInfo);
    if (*MinSlack > Slack)
        *MinSlack = Slack;

    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                    GnlTimingInfoArrivalFall (TimingInfo);
    if (*MinSlack > Slack)
        *MinSlack = Slack;

}
return (GNL_OK);
}

/*-----
/* GnlGetSequentialCritPathFromGnl
/*-----
/* Doc procedure GnlGetSequentialCritPathFromGnl :
- Getting the slack minimal attached to the inputs of sequential instances
of a given netlist (GNL).

/*-----
GNL_STATUS GnlGetSequentialCritPathFromGnl (GNL Gnl, LIBC_LIB      Lib,
                                              BLIST ListOfCritPaths,
                                              float MinSlack)
{
    GNL_VAR          Var;
    int             i, Rank;
    GNL_STATUS      GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float           Slack;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT   ComponentI;
    GNL             GnlCompoI;
    GNL_ASSOC       Assoc;
    GNL_CRITICAL_PATH CritPath;
    unsigned int     Key;

for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);
    if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
    {
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlGetSequentialCritPathFromGnl (GnlCompoI, Lib,
                                                ListOfCritPaths, MinSlack))
                return (GnlStatus);
            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        }
    }
}
```

```

        }
    }
    if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
        continue;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
    Assoc = GnlSequentialCompoInputAssoc (SeqCompo);
    if (!Assoc)
        continue;
    Var = GnlAssocActualPort (Assoc);
    if (!Var)
        continue;
    if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
        continue;
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    if (MinSlack == Slack)
    {
        if (GnlStatus = GnlGetCritPathFromInputOfSeqCell (Assoc, Lib, &CritPath,
                                                          (GNL_CRITICAL_PATH) NULL, RISE))
            return (GnlStatus);
        if (BListAddElt (ListOfCritPaths, CritPath))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        Slack = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
        if (MinSlack == Slack)
        {
            if (GnlStatus = GnlGetCritPathFromInputOfSeqCell (Assoc, Lib,
&CritPath,
                                                          (GNL_CRITICAL_PATH) NULL, FALL))
                return (GnlStatus);
            if (BListAddElt (ListOfCritPaths, CritPath))
                return (GNL_MEMORY_FULL);
        }
    }
    return (GNL_OK);
}

/*-----*/
/* GnlGetMinSlackFromOutputs */ */
/*-----*/
/* Doc procedure GnlGetMinSlackFromOutputs :
   - Getting the slack minimal attached to the primary outputs of a given
     netlist (GNL).

/*-----*/
GNL_STATUS GnlGetMinSlackFromOutputs (GNL Gnl, float *MinSlack)
{
    GNL_VAR          Var;

```

timecomp.c

```
int i, j, Rank;
GNL_STATUS GnlStatus;
GNL_TIMING_INFO TimingInfo;
float Slack;
BLIST BucketI;
unsigned int Key;

*MinSlack = MAX_FLOAT;
for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
                     (GnlVarDir (Var) != GNL_VAR_INOUT)) )
            continue;
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
            return (GnlStatus);

        Slack = GnlTimingInfoRequiredRise (TimingInfo) -
                GnlTimingInfoArrivalRise (TimingInfo);
        if (*MinSlack > Slack)
            *MinSlack = Slack;
        Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                GnlTimingInfoArrivalFall (TimingInfo);
        if (*MinSlack > Slack)
            *MinSlack = Slack;
    }
}
return (GNL_OK);
}

/*-----*/
GNL_STATUS GnlGetCombinationalCritPathFromGnl (GNL Gnl, LIBC_LIB Lib,
                                                BLIST ListOfCritPaths,
                                                float *MinSlackForCombPath,
                                                float *MinSlackForSeqPath)
{
    GNL_VAR Var;
    int i, j, Rank, IsCombPath, IsSeqPath;
    GNL_STATUS GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float MinSlack, Slack;
    GNL_CRITICAL_PATH CritPath;
    BLIST BucketI;
    unsigned int Key;

/* GnlGetMinSlackFromOutputs (Gnl,&MinSlack); */
for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
```

timecomp.c

```
Var = (GNL_VAR) BListElt (BucketI, j);
if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
             (GnlVarDir (Var) != GNL_VAR_INOUT)) )
    continue;

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                       &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

IsCombPath = IsSeqPath = 0;
if (GnlTimingInfoArrivalRise (TimingInfo) > GnlTimingInfoArrivalFall
(TimingInfo))
{
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    if (GnlStatus = GnlGetCritPathFromVar (Var, Lib, &CritPath,
                                           (GNL_CRITICAL_PATH) NULL, RISE,
                                           &IsCombPath, &IsSeqPath, 0, 0))
        return (GnlStatus);
}
else
{
    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
    if (GnlStatus = GnlGetCritPathFromVar (Var, Lib, &CritPath,
                                           (GNL_CRITICAL_PATH) NULL, FALL,
                                           &IsCombPath, &IsSeqPath, 0, 0))
        return (GnlStatus);
}

if (BListAddElt (ListOfCritPaths, CritPath))
    return (GNL_MEMORY_FULL);
if (IsCombPath)
    if (*MinSlackForCombPath > Slack)
        *MinSlackForCombPath = Slack;
if (IsSeqPath)
    if (*MinSlackForSeqPath > Slack)
        *MinSlackForSeqPath = Slack;

}

return (GNL_OK);
}

/*-----*/
/* GnlPrintHeadOfTimingReport */
/*-----*/
/* Doc procedure GnlPrintHeadOfTimingReport :
   - Printing global parameters used for timing analysis
*/
/*-----*/
GNL_STATUS GnlPrintHeadOfTimingReport (GNL      Gnl,
                                       LIBC_LIB  Lib)
{
    fprintf (stderr, " -----\\n");
}
```

timecomp.c

```
fprintf (stderr, " Report\t\t:ttiming\n");
fprintf (stderr, " max_number_paths\t:\t%d\n", GnlEnvPrintNbCritPath ());
fprintf (stderr, " Design\t\t: \t%s\n", GnlName (Gnl));
fprintf (stderr, " Date\t\t: \t");
GnlPrintDate (stderr);
fprintf (stderr, "\n");

fprintf (stderr, " Operating Conditions\t:\t%s\n",
         LibOperCondOcName (G_OperCond));
fprintf (stderr, " Library\t:\t%s\n", LibName (Lib));

fprintf (stderr, " Wire Loading Model\t:\t");
if (G_WireLoad)
    fprintf (stderr, "%s\n", LibWireLoadName (G_WireLoad));
else
    fprintf (stderr, "No Wire Load\n");
fprintf (stderr, " ----- \n");
fprintf (stderr, " \n");

}

/*-----
/* GnlGetCritPathFromNetwork */
/*-----
/* Doc procedure GnlGetCritPathFromNetwork :
   - Getting combinational critical path and Clock periode from a given
   Network (GNL_NETWORK) .
*/
/*-----*/
GNL_STATUS GnlGetCritPathFromNetwork (GNL_NETWORK      Nw,
                                      LIBC_LIB        Lib,
                                      int             MaxPaths,
                                      double          AreaWithoutNet,
                                      int             CellNumber,
                                      int             PrintClkDomain,
                                      int             PrintCritRegion)
{
    GNL           TopGnl;
    GNL_STATUS    GnlStatus;
    BLIST        ListOfCritPaths, ListOfSeqCritPaths;
    BLIST        AuxList, ListOfClocks;
    GNL_CRITICAL_PATH CritPath;
    int           NumberOfPaths, i, Size;
    float         MinSlack, ClockFreqValue, CombPathValue, MinSlackForSeqPath,
                  MinSlackForCombPath;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (CellNumber > MAX_CELL_NUMBER)
    {
        if (BListCreate (&G_PileOfComponent))
            return (GNL_MEMORY_FULL);

        if (BListCreate (&ListOfCritPaths))
            return (GNL_MEMORY_FULL);

        if (BListCreate (&ListOfSeqCritPaths))
```

timecomp.c

```
    return (GNL_MEMORY_FULL);

MinSlackForCombPath = MinSlackForSeqPath = MAX_FLOAT;
if ((GnlStatus = GnlGetCombinationalCritPathFromGnl (TopGnl, Lib,
                                                       ListOfCritPaths,
                                                       &MinSlackForCombPath,
                                                       &MinSlackForSeqPath)))
{
    return (GnlStatus);
}
NumberOfPaths = MaxPaths;

GnlPrintHeadOfTimingReport (TopGnl, Lib);
ClockFreqValue = CombPathValue = 0.0;

for (i=0; i < BListSize (ListOfCritPaths); i++)
{
    CritPath = (GNL_CRITICAL_PATH) BListElt (ListOfCritPaths, i);
    if (BListCreate (&AuxList))
        return (GNL_MEMORY_FULL);
    Size = BListSize (ListOfSeqCritPaths);
    GnlPrintNEquivalPathFromOneCritPath (CritPath, AuxList, &NumberOfPaths,
                                         0, &ClockFreqValue, &CombPathValue,
                                         MinSlackForSeqPath, MinSlackForCombPath,
                                         ListOfSeqCritPaths);

    if (BListSize (ListOfSeqCritPaths) != Size)
    {
        BListDelShift (ListOfCritPaths, i+1);
        i--;
    }

    BListQuickDelete (&AuxList);
    if (!NumberOfPaths)
        break;
}
BListDelete (&ListOfCritPaths, GnlFreeCritPath);

if (CombPathValue)
{
    fprintf (stderr, " -----\n");
    fprintf (stderr, " Combinational Path \t\t%.2f ns\n", CombPathValue);
    fprintf (stderr, " -----");
    fprintf (stderr, "\n\n");
}

MinSlack = MAX_FLOAT;
GnlGetMinSlackFromInputsOfSequentialInst (TopGnl, &MinSlack);
if (MinSlack < MinSlackForSeqPath)
{
    MinSlackForSeqPath = MinSlack;
    BListQuickDelete (&ListOfSeqCritPaths);
    if (BListCreate (&ListOfSeqCritPaths))
        return (GNL_MEMORY_FULL);
    if ((GnlStatus = GnlGetSequentialCritPathFromGnl (TopGnl, Lib,
                                                       ListOfSeqCritPaths, MinSlack)))
        return (GnlStatus);
}
```

timecomp.c

```
    }
    else if (MinSlack == MinSlackForSeqPath)
    {
        if ((GnlStatus = GnlGetSequentialCritPathFromGnl (TopGnl, Lib,
                                                        ListOfSeqCritPaths, MinSlack)))
            return (GnlStatus);
    }

    NumberOfPaths = MaxPaths;
    ClockFreqValue = 0.0;
    for (i=0; i < BListSize (ListOfSeqCritPaths); i++)
    {
        CritPath = (GNL_CRITICAL_PATH) BListElt (ListOfSeqCritPaths, i);
        if (BListCreate (&AuxList))
            return (GNL_MEMORY_FULL);
        GnlPrintNEquivalPathFromOneCritPath (CritPath, AuxList, &NumberOfPaths,
                                             1, &ClockFreqValue, &CombPathValue,
                                             MinSlackForSeqPath, MinSlackForCombPath,
                                             NULL);
        BListQuickDelete (&AuxList);
        if (!NumberOfPaths)
            break;
    }
    BListDelete (&ListOfSeqCritPaths, GnlFreeCritPath);
    if (ClockFreqValue)
    {
        fprintf (stderr, " -----\n");
        fprintf (stderr, " Clock Period \t\t%.2f ns\n", ClockFreqValue);
        fprintf (stderr, " Clock Frequency \t\t%.2f Mhz\n",
                1000 * 1/ClockFreqValue);
        fprintf (stderr, " -----");
        -----
    }
    BListQuickDelete (&ListOfCritPaths);
    if (PrintCritRegion)
        if (GnlGetEpsiCriticInstancesFromGnl (TopGnl, Lib,
                                              MinSlackForSeqPath, MinSlackForCombPath, AreaWithoutNet))
            return (GNL_MEMORY_FULL);
        BListQuickDelete (&G_PileOfComponent);
    }
    else
    {
        if (BListCreate (&ListOfClocks))
            return (GNL_MEMORY_FULL);
        if (GnlStatus = GnlGetClocksFromNetwork (Nw, ListOfClocks,
PrintClkDomain))
            return (GnlStatus);

        if (BListCreate (&G_PileOfComponent))
            return (GNL_MEMORY_FULL);
        GnlGetMinSlackFromOutputs (TopGnl, &MinSlackForCombPath);
        MinSlackForSeqPath = MAX_FLOAT;
        GnlGetMinSlackFromInputsOfSequentialInst (TopGnl, &MinSlackForSeqPath);
        BListQuickDelete (&G_PileOfComponent);
        if (GnlStatus = TimeExpandCritPathFromNetwork (Nw, Lib, MaxPaths,
                                                       ListOfClocks, AreaWithoutNet,
```

```

        MinSlackForCombPath,
        MinSlackForSeqPath,
        PrintCritRegion))

    return (GnlStatus);
BListDelete (&ListOfClocks, GnlFreeClock);
}
return (GNL_OK);
}

/*-----*/
/* TimeOptByResizing */
/*-----*/
GNL_STATUS TimeOptByResizing (GNL_NETWORK Nw,
                               LIBC_LIB      GnlLibc)
{
    double   AreaWithoutNet, AreaOfNets;
    int      CellNumber;
    GNL      TopGnl;
    int      Tag, i;
    LIBC_CELL   Buffer, CellI;
    BLIST    ListBuffers;
    float    InCapaBuffer, LimitCapaBuffer, MaxLimitCapaBuffer;

    TopGnl = GnlNetworkTopGnl(Nw);

    if (!GnlEnvRespectLibConstraints())
    {
        if (GnlLinkLib (Nw, TopGnl, GnlLibc))
            return (GNL_MEMORY_FULL);

        if (GnlSetListPathComponent (Nw))
            return (GNL_MEMORY_FULL);

        /* we resize the hash list of path component depending on the
        /* number of instances leading to a current Gnl. */
        if (GnlResizeHashListPathComponent (Nw))
            return (GNL_MEMORY_FULL);

        if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc))
            return (GNL_MEMORY_FULL);

        LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);

        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))
            return (GNL_MEMORY_FULL);
    }

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 0, &AreaWithoutNet, &AreaOfNets,
                               &CellNumber))
        return (GNL_MEMORY_FULL);

    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);
}

```

timecomp.c

```
if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))
    return (GNL_MEMORY_FULL);

fprintf (stderr, "\n");
fprintf (stderr,
        " Optimization By Resizing Critical Instances...\n");
fprintf (stderr, "\n");

Tag = GnlNetworkTag (Nw) + 1;

ListBuffers = GnlHLibCellsBuffers ((GNL_LIB)LibHook (GnlLibc));

MaxLimitCapaBuffer = 0.0;

Buffer = (LIBC_CELL) NULL;
for (i=0; i < BListSize (ListBuffers); i++)
{
    CellI = (LIBC_CELL) BListElt (ListBuffers, i);
    TimeGetInCapaAndLimitCapa (CellI, &InCapaBuffer, &LimitCapaBuffer,
GnlLibc);
    if (MaxLimitCapaBuffer < LimitCapaBuffer)
    {
        Buffer = CellI;
        MaxLimitCapaBuffer = LimitCapaBuffer;
    }
}

if (BListSize (ListBuffers) )
    if (TimeOptByResizingFromGnl (TopGnl, GnlLibc, 1, &Tag, ListBuffers,
Buffer))
        return (GNL_MEMORY_FULL);

if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))
    return (GNL_MEMORY_FULL);

if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))
    return (GNL_MEMORY_FULL);

if (TimeOptByResizingFromGnl (TopGnl, GnlLibc, 0, &Tag, ListBuffers,
Buffer))
    return (GNL_MEMORY_FULL);

CellNumber = 0;
if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
    &CellNumber))
    return (GNL_MEMORY_FULL);
fprintf (stderr, "\n");

return (GNL_OK);
}

/*-----
/* GnlMeetLibConstraints
*/
-----*/
GNL_STATUS GnlMeetLibConstraints (GNL_NETWORK Nw,
```

timecomp.c

```
LIBC_LIB      GnlLibc)
{
    double   AreaWithoutNet, AreaOfNets;
    int       CellNumber;
    GNL      TopGnl;

    TopGnl = GnlNetworkTopGnl (Nw) ;

    if (GnlLinkLib (Nw, TopGnl, GnlLibc))
        return (GNL_MEMORY_FULL);

    if (GnlSetListPathComponent (Nw) )
        return (GNL_MEMORY_FULL);

    /* we resize the hash list of path component depending on the      */
    /* number of instances leading to a current Gnl.                  */
    if (GnlResizeHashListPathComponent (Nw) )
        return (GNL_MEMORY_FULL);

    if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc) )
        return (GNL_MEMORY_FULL);

    LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);

    if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc) )
        return (GNL_MEMORY_FULL);

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
                                &CellNumber))
        return (GNL_MEMORY_FULL);

    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc) )
        return (GNL_MEMORY_FULL);

    if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc) )
        return (GNL_MEMORY_FULL);

    fprintf (stderr, "\n");
    fprintf (stderr,
             " Restructuring Netlist to meet Library Constraints...\n");
    fprintf (stderr, "\n");
    if (TimeLimitationFanoutFromNetwork (Nw, GnlLibc) )
        return (GNL_MEMORY_FULL);

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
                                &CellNumber))
        return (GNL_MEMORY_FULL);
    fprintf (stderr, "\n");

    return (GNL_OK);
}

/*-----*/
```

timecomp.c

```
/* TimeEstimate */  
/*-----*/  
GNL_STATUS TimeEstimate (GNL_NETWORK Nw,  
                         LIBC_LIB GnlLibc,  
                         int PrintClkDomain,  
                         int PrintCritRegion)  
{  
    double AreaWithoutNet, AreaOfNets;  
    int CellNumber;  
    GNL TopGnl;  
    int PrintData;  
  
    /* We call the construction of these data if:  
     * - we are in the ESTIMATION mode or  
     * - we are in another mode but we did not invoked the  
     *   control fanout.  
    */  
    if ((GnlEnvMode () == GNL_MODE_ESTIMATION) ||  
        (!GnlEnvRespectLibConstraints () && !GnlEnvPostOpt () &&  
         (GnlEnvMode () != GNL_MODE_POST_OPTIMIZATION) ) )  
    {  
        TopGnl = GnlNetworkTopGnl (Nw);  
  
        if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc))  
            return (GNL_MEMORY_FULL);  
  
        LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);  
  
        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))  
            return (GNL_MEMORY_FULL);  
    }  
  
    if (GnlEnvPostOpt () || (GnlEnvMode () == GNL_MODE_POST_OPTIMIZATION) )  
        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))  
            return (GNL_MEMORY_FULL);  
  
    CellNumber = 0;  
    PrintData = (!GnlEnvRespectLibConstraints () ||  
                (!GnlEnvPostOpt () &&  
                 (GnlEnvMode () != GNL_MODE_POST_OPTIMIZATION)) ||  
                (GnlEnvMode () == GNL_MODE_ESTIMATION) );  
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, PrintData, &AreaWithoutNet,  
                               &AreaOfNets, &CellNumber))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlGetCritPathFromNetwork (Nw, GnlLibc, GnlEnvPrintNbCritPath (),  
                                 AreaWithoutNet, CellNumber, PrintClkDomain,  
                                 PrintCritRegion))  
        return (GNL_MEMORY_FULL);
```

timecomp.c

```
if (GnlFreeTimingInfoFromNetwork (Nw) )
    return (GNL_MEMORY_FULL) ;

return (GNL_OK) ;
}

/* ----- EOF ----- */
```

timecomp.e

```
/*-----*/  
/*-----*/  
/* File: timecomp.e */  
/* Version: 1.1 */  
/* Modifications: - */  
/* Documentation: - */  
/*-----*/  
/* Description: */  
/*-----*/  
  
extern int GnlFloatEqual (float a, float b, int precision);  
  
extern float GnlMaxFloat (float a, float b);  
extern float GnlMinFloat (float a, float b);  
  
extern GNL_STATUS GnlGetHierarchycalInstName (char *InstanceName,  
                                              char **HierarchycalName);  
  
GNL_STATUS GnlComputeSetupHoldTimeFromSeqCompo (  
    GNL_SEQUENTIAL_COMPONENT SeqCompo,  
    char *PinInName,  
    GNL_TIMING_INFO TimingInfoOfPin,  
    float *TimeRise,  
    float *TimeFall,  
    LIBC_LIB Lib,  
    int WithRec);  
  
extern GNL_STATUS GnlTimingInfoCorrespToCurrentPathFromVar (GNL_VAR Var,  
    GNL_TIMING_INFO *TimingInfo,  
    unsigned int *Key,  
    int *Rank);  
  
extern GNL_STATUS GnlComputeCapaAndFanoutFromVar (GNL_VAR Var,  
    LIBC_LIB Lib,  
    int Tag,  
    int InoutIsIn,  
    int VarIsInout);  
  
extern GNL_STATUS GnlComputeCapaAndFanoutFromNetwork (GNL_NETWORK Nw,  
    LIBC_LIB Lib);  
  
extern GNL_ASSOC GnlGetAssocFromFormalPortAndHierBlock (GNL_VAR Formal,  
    GNL_USER_COMPONENT HierBlock,  
    GNL_VAR *Actual);  
  
extern GNL_STATUS GnlComputeArrivalTimeFromVar (GNL_VAR Var,  
    LIBC_LIB Lib,  
    int Tag,  
    int WithRec,  
    int InoutIsIn,  
    int VarIsInout);  
  
extern GNL_STATUS GnlComputeArrivalTimeFromNetwork (GNL_NETWORK Nw,  
    LIBC_LIB Lib);  
  
extern GNL_STATUS GnlComputeRequiredTimeFromVar (GNL_VAR Var,
```

```

        LIBC_LIB Lib,
        int      Tag,
        int      WithRec,
        int      InoutIsIn,
        int      VarIsInout);

extern GNL_STATUS GnlComputeRequiredTimeFromAssoc (GNL_ASSOC          Assoc,
                                                 float    *TimeRise,
                                                 float    *TimeFall,
                                                 LIBC_LIB Lib,
                                                 GNL_VAR Actual,
                                                 int      Tag,
                                                 int      WithRec);

extern GNL_STATUS GnlComputeRequiredTimeFromNetwork (GNL_NETWORK Nw,
                                                    LIBC_LIB Lib);

extern void GnlGetArrivalTimeRiseAndFallSwitchTimingSence (LIBC_PIN PinIn,
                                                          LIBC_PIN PinOut,
                                                          float DelayR, float DelayF,
                                                          float TimeRiseIn, float TimeFallIn,
                                                          float *TimeRise, float *TimeFall);

extern GNL_STATUS GnlGetCritPathFromNetwork (GNL_NETWORK      Nw,
                                             LIBC_LIB      Lib,
                                             int          MaxPaths,
                                             double       AreaWithoutNet,
                                             int          CellNumber,
                                             int          PrintClkDomain,
                                             int          PrintCritRegion);

extern GNL_ASSOC GnlGetAssocFromFormalPortAndHierBlock (GNL_VAR Formal,
                                                       GNL_USER_COMPONENT HierBlock,
                                                       GNL_VAR *Actual);

extern GNL_STATUS TimeGetFormalFromAssocAndActual (GNL_ASSOC Assoc,
                                                   GNL_VAR Actual, GNL_VAR *Formal);

extern GNL_STATUS TimeEstimate (GNL_NETWORK Nw, LIBC_LIB GnlLibc,
                               int PrintClkDomain,
                               int PrintCritRegion);

extern GNL_STATUS GnlMeetLibConstraints (GNL_NETWORK      Nw,
                                         LIBC_LIB      GnlLibc);

extern GNL_STATUS TimeOptByResizing (GNL_NETWORK      Nw,
                                     LIBC_LIB      GnlLibc);
/*-----*/
/*----- EOF -----*/

```

timecomp2.c

```
/*
 *-----*/
/* File: timecomp.c */
/* Version: 1.1 */
/* Modifications: - */
/* Documentation: - */
/* Description: */
/*-----*/
#include <stdio.h>

#ifndef MEMORY /* If one wants memory statistics for SUN */
#include <malloc.h>
#endif

#include "blist.h"
#include "gnl.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnlestim.h"
#include "time.h"
#include "bbdd.h"
#include "gnlmap.h"
#include "gnloption.h"

#include "blist.e"
#include "libutil.e"
#include "timeutil.e"
#include "timecomp.e"
#include "timepath.e"
#include "timefan.e"

#define MAX_CELL_NUMBER 30000
/*-----*/
/* EXTERN */
/*-----*/
extern GNL_ENV G_GnlEnv;
/*-----*/
/*----- Global Variable of Traversal of the Netlist -----*/
GNL_USER_COMPONENT G_CurrentComponent;
BLIST G_PileOfComponent;
/*-----*/
/*-----*/
int GnlFloatEqual (float a, float b, int precision)
{
    float f, puiss = 1.0;
    int i;

    for (i=0 ; i< precision ; i++)
        puiss *= 10;
```

timecomp2.c

```
if (a > b)
    f = (a - b) * puiss ;
else
    f = (b - a) * puiss;

i = (f < 1) ? 1 : 0;
return (i);
}

/*-----
float GnlMaxFloat (float a, float b)
{
    float Max;

    Max = a;
    if (Max < b)
        Max = b;

    return (Max);
}

/*-----
float GnlMinFloat (float a, float b)
{
    float Min;

    Min = a;
    if (Min > b)
        Min = b;

    return (Min);
}

/*-----
/*-----*/
/* GnlGetHierarchycalInstName */
/*-----*/
GNL_STATUS GnlGetHierarchycalInstName (char *InstanceName,
                                         char **HierarchycalName)
{
    char *Str1, *Str2, *InstName;
    int i;
    GNL_USER_COMPONENT     AuxUserCompo;

    InstName = NULL;
    if (BListSize (G_PileOfComponent))
    {
        AuxUserCompo = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent, 0);
        Str2 = GnlUserComponentInstName (AuxUserCompo);
        if (GnlStrCopy (Str2, &InstName))
            return (GNL_MEMORY_FULL);
        for (i=1; i < BListSize (G_PileOfComponent); i++)
        {
            Str1 = InstName;
            if (GnlStrAppendStrCopy (Str1, GnlEnvFlattenStr(), &InstName))
                return (GNL_MEMORY_FULL);
    }
}
```

timecomp2.c

```
free (Str1);
Str1 = InstName;
AuxUserCompo = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent, i);
Str2 = GnlUserComponentInstName (AuxUserCompo);
if (GnlStrAppendStrCopy (Str1, Str2, &InstName))
    return (GNL_MEMORY_FULL);
free (Str1);
}
if (InstanceName)
{
    Str1 = InstName;
    if (GnlStrAppendStrCopy (Str1, GnlEnvFlattenStr (), &InstName))
        return (GNL_MEMORY_FULL);
    free (Str1);
    Str1 = InstName;
    if (GnlStrAppendStrCopy (Str1, InstanceName, &InstName))
        return (GNL_MEMORY_FULL);
    free (Str1);
}
else
if (InstanceName)
    if (GnlStrCopy (InstanceName, &InstName))
        return (GNL_MEMORY_FULL);
*HierarchycalName = InstName;
return (GNL_OK);
}

/*
 * SetGnlCritPathInstNameFromUserCompo
 */
GNL_STATUS SetGnlCritPathInstNameFromUserCompo (GNL_CRITICAL_PATH CritPath,
                                                char *InstanceName)
{
    char *HierarchycalName;

    if (!CritPath || !InstanceName)
        return (GNL_OK);
    if (GnlGetHierarchycalInstName (InstanceName, &HierarchycalName))
        return (GNL_MEMORY_FULL);

    SetGnlCritPathInstName (CritPath, HierarchycalName);
    return (GNL_OK);
}
/*
 * GnlPrintCritPath
 */
void GnlPrintCritPath (BLIST ListCritPath, float *ClockFreqValue,
                      float *CombPathValue)
{
    GNL_CRITICAL_PATH      CritPath;
    LIBC_CELL              Cell;
    GNL_COMPONENT          Compo;
    GNL_VAR                Var, Formal, StartVar, EndVar;
    int                   i, IsSequential;
    GNL_ASSOC              Assoc;
```

```

CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath,
                                         BListSize(ListCritPath) -1);
StartVar = GnlCritPathCriticalVar (CritPath);
CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, 0);
EndVar = GnlCritPathCriticalVar (CritPath);

IsSequential = (!StartVar || !EndVar);

fprintf (stderr, " Point\t\t\t\t\tFanout\tIncr\tPath\n");
fprintf (stderr, " -----
-----\n");

for (i= BListSize(ListCritPath) -1; i >= 0; i--)
{
    CritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, i);
    if (!GnlCritPathPredecessors (CritPath) ||
        !GnlCritPathSuccessors (CritPath))
    {
        Var = GnlCritPathCriticalVar (CritPath);
        if (Var)
        {
            if (!GnlCritPathPredecessors (CritPath))
            {
                /* the input external delay is depending of constraint timing */
                /* For the moment it is 0.0 (later we have to do that      */
                fprintf (stderr, " input external delay\t\t\tt0.0\t0.0 \n");
            }
            fprintf (stderr, "%s ", GnlVarName (Var));

            if (!GnlCritPathPredecessors (CritPath))
                fprintf (stderr, "(in)\n\t\t\t\t\t\t\t");
            else
                fprintf (stderr, "(out)\n\t\t\t\t\t\t\t");
            fprintf (stderr, "% .2f\t% .2f", GnlCritPathIncr (CritPath),
                     GnlCritPathArrTime (CritPath));
            if (GnlCritPathKindOfTime (CritPath) == RISE)
                fprintf (stderr, " Rising\n");
            else
                fprintf (stderr, " Falling\n");
        }
    }
}

Assoc = GnlCritPathCriticalAssoc (CritPath);
if (Assoc)
{
    Compo = GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);
    Formal = GnlAssocFormalPort (Assoc);
    GnlGetCellFromGnlCompo (Compo, &Cell);
    if (Cell)
    {
        fprintf (stderr, "%s", GnlCritPathInstName (CritPath));
        fprintf (stderr, "/%s %s", (char *) Formal, GnlVarName (Var));
        fprintf (stderr, " (%s)\n\t\t\t\t\t\t", LibCellName (Cell));
        if (IsSequential && i==0)
            fprintf (stderr, "\t0.00\t% .2f", GnlCritPathArrTime (CritPath));
    }
}

```

timecomp2.c

```

if (NumberOfPaths <= 0)
    return (GNL_OK);

if (BListAddElt (ListCritPath, CritPath))
    return (GNL_MEMORY_FULL);

if (GnlCritPathPredecessors (CritPath))
{
    for (i=0; i < BListSize (GnlCritPathPredecessors (CritPath)); i++)
    {
        Predecessor = (GNL_CRITICAL_PATH) BListElt (
            GnlCritPathPredecessors (CritPath), i);

        GnlPrintNEquivPathFromOneCritPath (Predecessor, ListCritPath,
            NumberOfPaths, IsSequential, ClockFreqValue,
            CombPathValue, MinSlackForSeqPath,
            MinSlackForCombPath,
            ListPathClockFreq);
        BListDelShift (ListCritPath, BListSize (ListCritPath));
    }
}
else
{
    if (*NumberOfPaths > 0)
    {
        EndCritPath = (GNL_CRITICAL_PATH) BListElt (ListCritPath, 0);
        Slack = GnlCritPathReqTime(EndCritPath) -
        GnlCritPathArrTime(EndCritPath);
        StartVar = GnlCritPathCriticalVar (CritPath);
        EndVar = GnlCritPathCriticalVar (EndCritPath);
        if (!IsSequential)
        {
            if (StartVar && GnlFloatEqual (MinSlackForCombPath, Slack, 4))
            {
                GnlPrintCritPath (ListCritPath, ClockFreqValue, CombPathValue);
                (*NumberOfPaths)--;
            }
            else
            {
                if (GnlFloatEqual (MinSlackForSeqPath, Slack, 4))
                    if (BListAddElt (ListPathClockFreq, BListElt (ListCritPath, 0)))
                        return (GNL_MEMORY_FULL);
                }
                return (GNL_OK);
            }
        else
        {
            if (!StartVar || !EndVar)
            {
                GnlPrintCritPath (ListCritPath, ClockFreqValue, CombPathValue);
                (*NumberOfPaths)--;
            }
        }
    }
}
return (GNL_OK);

```

timecomp2.c

```
}

/*-----*/
/*-----*/
/* GnlComparePathToCurrentPath */
/*-----*/
int GnlComparePathToCurrentPath (GNL_PATH_COMPONENT Path,
                                BLIST CurrentPath)
{
    int i;
    GNL_PATH_COMPONENT Previous;
    GNL_USER_COMPONENT CompoI;

    if (!Path)
        return (0);

    CompoI = GnlPathComponentComponent (Path);
    Previous = GnlPathComponentPrevious (Path);
    for (i= BListSize (CurrentPath)-1; i >=0; i--)
    {
        if (CompoI != (GNL_USER_COMPONENT) BListElt (CurrentPath, i))
            return (0);
        if (Previous && (i == 0))
            return (0);
        if (!Previous && (i > 0))
            return (0);
        if (Previous)
        {
            CompoI = GnlPathComponentComponent (Previous);
            Previous = GnlPathComponentPrevious (Previous);
        }
    }

    return (1);
}
/*-----*/
/* TimingInfoCorrespToCurrentPathFromVar () */
/*-----*/
GNL_STATUS GnlTimingInfoCorrespToCurrentPathFromVar (
    GNL_VAR Var,
    GNL_TIMING_INFO *TimingInfo,
    unsigned int *Key,
    int *Rank)
{
    GNL_STATUS GnlStatus;
    BLIST ListTimingInfo, ListPaths;
    GNL CurrentGnl;
    GNL_USER_COMPONENT CurrentComponent;
    char *InstanceName;
    BLIST SubList, SubListForTiming;

    *Rank = 0;
    if (BListSize (G_PileOfComponent))
    {
        /* we take the last element of the pile */
    }
}
```

timecomp2.c

```
/* e.g the current instance. */  
CurrentComponent = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,  
                                         BListSize (G_PileOfComponent)-1);  
CurrentGnl = GnlUserComponentGnlDef (CurrentComponent);  
  
/* we access the hash list. */  
ListPaths = GnlListPathComponent (CurrentGnl);  
InstanceName = GnlUserComponentInstName (CurrentComponent);  
*Key = KeyOfName (InstanceName, BListSize (ListPaths));  
SubList = (BLIST)BListElt (ListPaths, *Key);  
  
*Rank = BListMemberOfList (SubList, G_PileOfComponent,  
                           GnlComparePathToCurrentPath);  
  
ListTimingInfo = (BLIST) GnlVarTraversalInfoHook (Var);  
if (!ListTimingInfo)  
{  
    if (BListCreateWithSize (BListSize (ListPaths), &ListTimingInfo))  
        return (GNL_MEMORY_FULL);  
    BSize (ListTimingInfo) = BListSize (ListPaths);  
    SetGnlVarTraversalInfoHook (Var, (void*) ListTimingInfo);  
  
    if (BListCreateWithSize (BListSize (SubList), &SubListForTiming))  
        return (GNL_MEMORY_FULL);  
    BSize (SubListForTiming) = BListSize (SubList);  
    BListElt (ListTimingInfo, *Key) = (int) SubListForTiming;  
    *TimingInfo = (GNL_TIMING_INFO) NULL;  
}  
else  
{  
    SubListForTiming = (BLIST) BListElt (ListTimingInfo, *Key);  
    if (!SubListForTiming)  
    {  
        if (BListCreateWithSize (BListSize (SubList), &SubListForTiming))  
            return (GNL_MEMORY_FULL);  
        BSize (SubListForTiming) = BListSize (SubList);  
        BListElt (ListTimingInfo, *Key) = (int) SubListForTiming;  
    }  
    *TimingInfo = (GNL_TIMING_INFO) BListElt (SubListForTiming, *Rank-1);  
}  
}  
else  
    *TimingInfo = (GNL_TIMING_INFO) GnlVarTraversalInfoHook (Var);  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* GnlComputeCapaAndFanoutFromSeqCell */  
/*-----*/  
/* Doc procedure GnlComputeCapaAndFanoutFromSeqCell :  
   - For a given GNL_ASSOC (Assoc) :  
     - Compute the output capacitance "Capa" and the fanout "Fanout" of  
       the net "GnlAssocActualPort (Assoc)".  
   - it does not include the wire capacitance.  
   - the GNL_ASSOC is a connector of a Sequential component.  
*/
```

timecomp2.c

```
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromSeqCell (GNL_ASSOC      Assoc,
                                                float        *Capa,
                                                int         *Fanout,
                                                LIBC_LIB Lib)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinIn;
    GNL_ASSOC     AssocI;
    char          *Formal;
    LIBC_KFATOR   KF;
    float         AuxCapa;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E )
        return (GNL_OK);

    AuxCapa = LibPinCapa (PinIn);
    if (AuxCapa == 0.0)
        AuxCapa = LibDefaultInputPinCap (Lib);
    KF = LibKFactor (Lib);
    AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                           LibKFactorPinCap(KF) [1],
                           LibKFactorPinCap(KF) [2]);
    *Capa += AuxCapa;
    (*Fanout)++;
    return (GNL_OK);

}
/*-----*/
/* GnlComputeCapaAndFanoutFrom3State                               */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFrom3State :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the output capacitance "Capa" and the fanout "Fanout" of
       the net "GnlAssocActualPort (Assoc)".
     - it does not include the wire capacitance.
     - the GNL_ASSOC is a connector of a 3state component.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFrom3State (GNL_ASSOC      Assoc,
                                                float        *Capa,
                                                int         *Fanout,
                                                LIBC_LIB Lib)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinIn;
    GNL_ASSOC     AssocI;
    char          *Formal;
    LIBC_KFATOR   KF;
```

timecomp2.c

```
float AuxCapa;
GNL_TRISTATE_COMPONENT TriStateCompo;

TriStateCompo =
(GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Cell = GnlTriStateCompoInfoCell (TriStateCompo);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E )
    return (GNL_OK);

AuxCapa = LibPinCapa (PinIn);
if (AuxCapa == 0.0)
    AuxCapa = LibDefaultInputPinCap (Lib);
KF = LibKFactor (Lib);
AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF)[0],
                      LibKFactorPinCap(KF)[1],
                      LibKFactorPinCap(KF)[2]);
*Capa += AuxCapa;
(*Fanout)++;
return (GNL_OK);

}
/*-----*/
GNL_STATUS TimeGetFormalFromAssocAndActual (GNL_ASSOC Assoc, GNL_VAR Actual,
                                             GNL_VAR *Formal)
{
    GNL_VAR AuxActual, AuxFormal, IndexFormalVar, SplitActualJ;
    GNL_USER_COMPONENT UserCompo;
    GNL GnlDefCompo;
    char *IndexFormalName;
    int i, j, Index, LeftFormIndex, RightFormIndex;
    BLIST ListSplitActuals;
    GNL_STATUS GnlStatus;

    *Formal = (GNL_VAR) NULL;
    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    GnlDefCompo = GnlUserComponentGnlDef (UserCompo);

    AuxActual = GnlAssocActualPort (Assoc);
    AuxFormal = GnlAssocFormalPort (Assoc);

    if (GnlVarIsVar (AuxActual))
    {
        *Formal = AuxFormal;
        return (GNL_OK);
    }
    else
    {
        ListSplitActuals = GnlNodeSons ((GNL_NODE) AuxActual);
        LeftFormIndex = GnlVarMsb (AuxFormal);
        RightFormIndex = GnlVarLsb (AuxFormal) ;

        if (LeftFormIndex < RightFormIndex)
            Index = LeftFormIndex-1;
```

timecomp2.c

```
else
    Index = LeftFormIndex+1;

for (j=0; j<BListSize (ListSplitActuals); j++)
{
    if (LeftFormIndex < RightFormIndex)
        Index++;
    else
        Index--;
    SplitActualJ = (GNL_VAR) BListElt (ListSplitActuals, j);
    if (SplitActualJ == Actual)
    {
        if (GnlVarIndexName (AuxFormal, Index, &IndexFormalName))
            return (GNL_MEMORY_FULL);

        if ((GnlStatus = GnlGetVarFromName (GnlDefCompo,
                                            IndexFormalName, &IndexFormalVar)))
        {
            free (IndexFormalName);
            if (GnlStatus == GNL_VAR_NOT_EXISTS)
            {
                fprintf (stderr, " ERROR: cannot find var\n");
                return (GNL_VAR_NOT_EXISTS);
            }
            return (GNL_MEMORY_FULL);
        }
        *Formal = IndexFormalVar;
        return (GNL_OK);
    }
}

return (GNL_OK);
}

/*
 */
/* GnlComputeCapaAndFanoutFromUserCompo */
/* Doc procedure GnlComputeCapaAndFanoutFromUserCompo :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the output capacitance "Capa" and the fanout "Fanout" of
       the net "GnlAssocActualPort (Assoc)" .
     - it does not include the wire capacitance.
     - the GNL_ASSOC is a connector of a UserComponent
*/
GNL_STATUS GnlComputeCapaAndFanoutFromUserCompo (GNL_ASSOC Assoc,
                                                 float      *Capa,
                                                 int        *Fanout,
                                                 LIBC_LIB Lib,
                                                 GNL_VAR   Actual,
                                                 int        Tag)
{
    GNL_VAR          Var, Formal, Vari, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
```

timecomp2.c

```
LIBC_CELL      Cell;
LIBC_PIN       Pin;
GNL_STATUS     GnlStatus;
LIBC_KFATOR   KF;
float          AuxCapa;
int            Rank, i;
GNL_ASSOC      AssocI;
BLIST          Interface;
unsigned int   Key;
int            AuxVarIsInout;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Var = GnlAssocActualPort (Assoc);
Interface = GnlUserComponentInterface (UserCompo);
Formal = GnlAssocFormalPort (Assoc);
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
        if (! (Pin = LibGetPinFromNameAndCell (Cell, (char *) Formal)))
            return (GNL_OK);
        if (LibPinDirection(Pin) == OUTPUT_E)
            return (GNL_OK);

        AuxCapa = LibPinCapa (Pin);
        if (AuxCapa == 0.0)
            AuxCapa = LibDefaultInputPinCap (Lib);
        KF = LibKFactor (Lib);
        AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                               LibKFactorPinCap(KF) [1],
                               LibKFactorPinCap(KF) [2]);
        *Capa += AuxCapa;
        (*Fanout)++;
        return (GNL_OK);
    }
    else
        /* It is a black box without any definition so we stop */
        /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
    return (GnlStatus);
```

timecomp2.c

```
AuxVarIsInout = (GnlVarDir (Actual) == GNL_VAR_INOUT);

if (GnlComputeCapaAndFanoutFromVar (AuxFormal, Lib, Tag, 1,
AuxVarIsInout))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal,
                                              &TimingInfo, &Key, &Rank))
    return (GNL_MEMORY_FULL);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*Capa += GnlTimingInfoCapa (TimingInfo);
*Fanout += GnlTimingInfoFanout (TimingInfo);

return (GNL_OK);
}

/*-----*/
/* GnlComputeCapaAndFanoutFromAssoc */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromAssoc :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the output capacitance "Capa" and the fanout "Fanout" of
       the net "GnlAssocActualPort (Assoc)".
     - it does not include the wire capacitance.
     - the GNL_ASSOC is a connector of a (user component "hierrachycal block or
generic cell", Sequential component "Flops or Latches", TriStates).
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromAssoc (GNL_ASSOC      Assoc,
                                              float        *Capa,
                                              int          *Fanout,
                                              LIBC_LIB     Lib,
                                              GNL_VAR     Actual,
                                              int          Tag)
{
    GNL_STATUS      GnlStatus;
    int            Rank;
    GNL_COMPONENT  GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            if (GnlStatus = GnlComputeCapaAndFanoutFromSeqCell (Assoc, Capa,
                                                               Fanout, Lib))
                return (GnlStatus);
            break;

        case GNL_USER_COMPO:
            if (GnlComputeCapaAndFanoutFromUserCompo (Assoc, Capa, Fanout,
                                                       Lib, Actual, Tag))
                return (GNL_MEMORY_FULL);
            break;
    }
}
```

timecomp2.c

```
case GNL_TRISTATE_COMPO:
    if (GnlStatus = GnlComputeCapaAndFanoutFrom3State (Assoc,
                                                       Capa, Fanout, Lib))
        return (GnlStatus);
    break;

case GNL_MACRO_COMPO:
    break;

default:
    GnlError (12 /* Unknown component */);
    break;
}

return (GNL_OK);
}
/*-----
/* GnlGetVarFromFormalPortAndHierBlock
 *-----
GNL_ASSOC GnlGetAssocFromFormalPortAndHierBlock (GNL_VAR Formal,
                                                GNL_USER_COMPONENT HierBlock,
                                                GNL_VAR *Actual)
{
    BLIST           Interface, ListSplitActuals;
    GNL_VAR     AuxFormal, AuxActual, SplitActualJ, IndexFormalVar;
    int          i, j, LeftFormIndex, RightFormIndex, Index;
    GNL_ASSOC AssocI;
    char         *IndexFormalName;
    GNL_STATUS   GnlStatus;

    *Actual = (GNL_VAR) NULL;
    if (!Formal || !HierBlock)
        return ((GNL_ASSOC) NULL);

    if (!GnlUserComponentGnlDef (HierBlock))
        return ((GNL_ASSOC) NULL);

    Interface = GnlUserComponentInterface (HierBlock);
    for (i=0; i < BlistSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        AuxFormal = GnlAssocFormalPort (AssocI);
        AuxActual = GnlAssocActualPort (AssocI);
        if (Formal == AuxFormal)
        {
            *Actual = AuxActual;
            if (!GnlVarIsVar (AuxActual))
                return ((GNL_ASSOC) NULL);
            return (AssocI);
        }

        if (!GnlVarIsVar (AuxActual))
        {
            LeftFormIndex = GnlVarMsb (AuxFormal);
            RightFormIndex = GnlVarLsb (AuxFormal);

            if (LeftFormIndex < RightFormIndex)
```

timecomp2.c

```
    Index = LeftFormIndex-1;
else
    Index = LeftFormIndex+1;

ListSplitActuals = GnlNodeSons ((GNL_NODE) AuxActual);
for (j=0; j < BListSize (ListSplitActuals); j++)
{
    if (LeftFormIndex < RightFormIndex)
        Index++;
    else
        Index--;
    SplitActualJ = (GNL_VAR) BListElt (ListSplitActuals, j);
    GnlVarIndexName (AuxFormal, Index, &IndexFormalName);
    if ((GnlStatus
=GnlGetVarFromName (GnlUserComponentGnlDef (HierBlock),
                    IndexFormalName, &IndexFormalVar)))
    {
        if (GnlStatus == GNL_VAR_NOT_EXISTS)
        {
            free (IndexFormalName);
            continue;
        }
        return ((GNL_ASSOC) NULL);
    }
    free (IndexFormalName);

    if (Formal == IndexFormalVar)
    {
        *Actual = SplitActualJ;
        return (AssocI);
    }
}
return ((GNL_ASSOC) NULL);
}/*-----*/
/* GnlComputeCapaAndFanoutFromVar */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromVar :
   - For a given GNL_VAR (Var) :
      - Compute the output capacitance and the fanout of
        the net "Var".
   - it does not include the wire capacitance.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromVar (GNL_VAR      Var,
                                             LIBC_LIB     Lib,
                                             int          Tag,
                                             int          InoutIsIn,
                                             int          VarIsInout)
{
    GNL_VAR          AuxVar, Actual;
    GNL_ASSOC        AuxAssocVar;
    int              i, Fanout, Rank;
    GNL_STATUS       GnlStatus;
```

timecomp2.c

```
BLIST          AssocDest, LeftVarAssigneds, ListTimingInfo,
SubList;
GNL_TIMING_INFO TimingInfo, AuxTimingInfo;
GNL_USER_COMPONENT InstOfGnl;
float           Capa;
unsigned int    Key;
int             AuxVarIsInout;

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                           &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if(TimingInfo)
{
    if ((GnlVarDir (Var) == GNL_VAR_INOUT || VarIsInout) && InoutIsIn)
    {
        AuxTimingInfo = GnlTimingInfoHook (TimingInfo);
        if (!AuxTimingInfo)
        {
            if (GnlCreateTimingInfo (&AuxTimingInfo))
                return (GNL_MEMORY_FULL);
            SetGnlTimingInfoHook (TimingInfo, AuxTimingInfo);
        }
        else
            return (GNL_OK);
        TimingInfo = GnlTimingInfoHook (TimingInfo);
    }
}
else
{
    if (GnlCreateTimingInfo (&TimingInfo))
        return (GNL_MEMORY_FULL);

    if (!BListSize (G_PileOfComponent))
        SetGnlVarTraversalInfoHook (Var, (void*) TimingInfo);
    else
    {
        ListTimingInfo = (BLIST) GnlVarTraversalInfoHook (Var);
        SubList = (BLIST) BListElt (ListTimingInfo, Key);

        BListElt (SubList, Rank-1) = (int) TimingInfo;
    }
    if ((GnlVarDir (Var) == GNL_VAR_INOUT || VarIsInout) && InoutIsIn)
    {
        if (GnlCreateTimingInfo (&AuxTimingInfo))
            return (GNL_MEMORY_FULL);
        SetGnlTimingInfoHook (TimingInfo, AuxTimingInfo);
        TimingInfo = GnlTimingInfoHook (TimingInfo);
    }
}
if (GnlTimingInfoTag (TimingInfo) == Tag)
    return (GNL_OK);
SetGnlTimingInfoTag (TimingInfo, Tag);
Capa = 0.0;
Fanout = 0;
```

timecomp2.c

```
if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
    (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn) )
{
    if (!BListSize (G_PileOfComponent))
    {
        /* Take the constraint from user (for later) */
        /*if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
            (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn)) */
        {
            SetGnlTimingInfoCapa (TimingInfo, Capa);
            SetGnlTimingInfoFanout (TimingInfo, 1);
            return (GNL_OK);
        }
    }
else
{
    InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                       &Actual);
    if (!AuxAssocVar)
    {
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }
    if (GnlComputeCapaAndFanoutFromVar (Actual, Lib, Tag, 0, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar (Actual, &AuxTimingInfo,
                                                   &Key, &Rank))
        return (GNL_MEMORY_FULL);

    Capa += GnlTimingInfoCapa (AuxTimingInfo);
    Fanout += GnlTimingInfoFanout (AuxTimingInfo);

    if (BListAddElt (G_PileOfComponent, InstOfGnl))
        return (GNL_MEMORY_FULL);
    }

if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (LeftVarAssigneds, i);
    if (GnlComputeCapaAndFanoutFromVar (AuxVar, Lib, Tag, 0, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar, &AuxTimingInfo,
&Key,
&Rank))
        return (GNL_MEMORY_FULL);
```

timecomp2.c

```
Capa += GnlTimingInfoCapa (AuxTimingInfo);
Fanout += GnlTimingInfoFanout (AuxTimingInfo);
}
BListQuickDelete (&LeftVarAssigneds);

for (i=0; i < BListSize (AssocDest); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocDest, i);
    if ((GnlStatus = GnlComputeCapaAndFanoutFromAssoc (AuxAssocVar, &Capa,
                                                    &Fanout, Lib, Var, Tag)))
        return (GnlStatus);
}
BListQuickDelete (&AssocDest);

SetGnlTimingInfoCapa (TimingInfo, Capa);
SetGnlTimingInfoFanout (TimingInfo, Fanout);

return (GNL_OK);
}

/*-----*/
/* GnlComputeCapaAndFanoutFromGnl */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromGnl :
   - For a given GNL (Gnl) :
      - Hierarchically compute the output capacitance and the fanout on each
        GNL_VAR.
*/
/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromGnl (GNL          Gnl,
                                             LIBC_LIB     Lib,
                                             int          Tag)
{
    GNL_VAR          Var;
    int             i, j, J;
    GNL_STATUS      GnlStatus;
    GNL            GnlCompoI;
    GNL_COMPONENT   ComponentI;
    BLIST          Components, BucketI;

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
/*      if (StringIdentical (GnlName (GnlCompoI), "cat_register_block-rtl"))
            J=j; */

            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlComputeCapaAndFanoutFromGnl (GnlCompoI, Lib, Tag))
                return (GnlStatus);
            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        }
    }
}
```

timecomp2.c

```
    }

}

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BlistElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;

        /*      if (StringIdentical (GnlVarName (Var), "$M23")
               || StringIdentical (GnlVarName (Var),
"u_cat_register_block.reset_wdt"))
            J=j; */

        if (GnlComputeCapaAndFanoutFromVar (Var, Lib, Tag, 1, 0))
            return (GNL_MEMORY_FULL);
        if (GnlVarDir (Var) == GNL_VAR_INOUT)
            if (GnlComputeCapaAndFanoutFromVar (Var, Lib, Tag, 0, 0))
                return (GNL_MEMORY_FULL);
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlComputeCapaAndFanoutFromNetwork */
/*-----*/
/* Doc procedure GnlComputeCapaAndFanoutFromNetwork :
   - For a given Network (GNL_NETWORK) :
     - Hierarchicaly compute the output capacitance and the fanout on each
       GNL_VAR.

/*-----*/
GNL_STATUS GnlComputeCapaAndFanoutFromNetwork (GNL_NETWORK Nw,
                                              LIBC_LIB     Lib)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    int          Tag;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
    Tag = GnlNetworkTag (Nw);
    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlComputeCapaAndFanoutFromGnl (TopGnl, Lib, Tag)))
        return (GnlStatus);

    BListQuickDelete (&G_PileOfComponent);
    return (GNL_OK);
}
```

timecomp2.c

```
}

/*-----*/
void GnlGetArrivalTimeRiseAndFallSwitchTimingSence (LIBC_PIN PinIn, LIBC_PIN
PinOut,
          float DelayR, float DelayF,
          float TimeRiseIn, float TimeFallIn,
          float *TimeRise, float *TimeFall)
{

    LIBC_TIMING           Timing;

    *TimeRise = *TimeFall = 0.0;
    if (!DelayR && !DelayF)
        return;

    Timing = LibGetArcTiming (PinIn, PinOut);
    if (!Timing)
        return;

    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *TimeRise = DelayR + TimeRiseIn;
            *TimeFall = DelayF + TimeFallIn;
            break;

        case NEGATIVE_UNATE_E:
            *TimeRise = DelayR + TimeFallIn;
            *TimeFall = DelayF + TimeRiseIn;
            break;

        case NON_UNATE_E:
            *TimeRise = DelayR + TimeRiseIn;
            if (TimeRiseIn < TimeFallIn)
                *TimeRise = DelayR + TimeFallIn;

            *TimeFall = DelayF + TimeFallIn;
            if (TimeFallIn < TimeRiseIn)
                *TimeFall = DelayF + TimeRiseIn;
            break;
    }

}

/*-----*/
/* GnlComputeArrivalTimeAtOutputOfCombCell */ */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeAtOutputOfCombCell :
   - For a given a GNL_USER_COMPONENT (UserCompo) generic cell and
     GNL_ASSOC (AssocOutput) (output GNL_ASSOC of UserCompo) :
       - Compute the arrival time (falling and rising)
       - Compute the transition(falling and rising)
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeAtOutputOfCombCell(GNL_USER_COMPONENT
UserCompo,
                                                 GNL_ASSOC AssocOutput,
```

timecomp2.c

```
float *MaxTimeRise,
float *MaxTimeFall,
float *MaxOutTransRise,
float *MaxOutTransFall,
LIBC_LIB     Lib,
int        Tag,
int        WithRec)

{

    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR       Var, Vari;
    GNL_TIMING_INFO TimingI;
    float         InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall, Time, MaxTime;
    float         Length, WireCapa, WireResistance, OutCapa;
    BLIST         Interface;
    GNL_STATUS    GnlStatus;
    unsigned int   Key;

    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
    Var = GnlAssocActualPort (AssocOutput);
    Formal = (char *) GnlAssocFormalPort (AssocOutput);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E)
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlUserComponentInterface (UserCompo);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        Vari = GnlAssocActualPort (AssocI);
        if (GnlVarIsVar (Vari))
            if (GnlVarIsVdd (Vari) || GnlVarIsVss (Vari))
                continue;

        Formal = (char *) GnlAssocFormalPort (AssocI);
        if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
            continue;
        if (LibPinDirection (PinIn) != INPUT_E)
            continue;
        if (!LibGetArcTiming (PinIn, PinOut))
            continue;
    }
}
```

timecomp2.c

```
if (WithRec)
if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
    return (GNL_MEMORY_FULL);
if (GnlVarDir (VarI) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);

InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
    Fanout, Cell, PinIn, PinOut,
    &DelayR, &OutTransR,
    &DelayF, &OutTransF);
GnlGetArrivalTimeRiseAndFallSwitchTimingsence (PinIn, PinOut,
    DelayR, DelayF,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),
    &TimeRise, &TimeFall);

MaxTime = *MaxTimeRise;
if (MaxTime < *MaxTimeFall)
    MaxTime = *MaxTimeFall;
Time = TimeRise;
if (Time < TimeFall)
    Time = TimeFall;

if (MaxTime < Time)
{
    *MaxTimeRise = TimeRise;
    *MaxTimeFall = TimeFall;
    *MaxOutTransRise = OutTransR;
    *MaxOutTransFall = OutTransF;
}
}
return (GNL_OK);
}

/*-----
/* GnlComputeArrivalTimeFrom3State */ */
/*-----
/* Doc procedure GnlComputeArrivalTimeFrom3State :
- For a given GNL_ASSOC (Assoc) :
    - Compute the arrival time (falling and rising)
    - Compute the transition(falling and rising)
on the net "GnlAssocActualPort (Assoc)".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFrom3State (GNL_ASSOC Assoc,
    float *MaxTimeRise,
    float *MaxTimeFall,
    float *MaxOutTransRise,
    float *MaxOutTransFall,
    LIBC_LIB Lib,
    int Tag,
```

timecomp2.c

```
int WithRec)
{
    LIBC_CELL Cell;
    LIBC_PIN PinOut, PinIn;
    int i, Fanout, Rank;
    GNL_ASSOC AssocI;
    char *Formal;
    GNL_VAR Var, Vari;
    GNL_TIMING_INFO TimingI;
    float InTransRise, InTransFall, OutTransR, OutTransF,
          DelayR, DelayF, TimeRise, TimeFall, MaxTime, Time;
    float Length, WireCapa, WireResistance, OutCapa;
    BLIST Interface;
    GNL_TRISTATE_COMPONENT TriStateCompo;
    GNL_STATUS GnlStatus;
    unsigned int Key;

    TriStateCompo = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E)
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
&TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlTriStateInterface (TriStateCompo);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        if (!AssocI)
            continue;
        Vari = GnlAssocActualPort (AssocI);

        if (!Vari)
            continue;

        if (GnlVarIsVar (Vari))
            if (GnlVarIsVdd (Vari) || GnlVarIsVss (Vari))
                continue;

        if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
            continue;
    }
}
```

timecomp2.c

```
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if (WithRec)
if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
    return (GnlStatus);
if (GnlVarDir (VarI) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);

InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
    Fanout, Cell, PinIn, PinOut,
    &DelayR, &OutTransR,
    &DelayF, &OutTransF);
GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
    DelayR, DelayF,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),
    &TimeRise, &TimeFall);

MaxTime = *MaxTimeRise;
if (MaxTime < *MaxTimeFall)
    MaxTime = *MaxTimeFall;
Time = TimeRise;
if (Time < TimeFall)
    Time = TimeFall;

if (MaxTime < Time)
{
    *MaxTimeRise = TimeRise;
    *MaxTimeFall = TimeFall;
    *MaxOutTransRise = OutTransR;
    *MaxOutTransFall = OutTransF;
}
}
return (GNL_OK);

}
/*-----*/
/* GnlComputeArrivalTimeFromSeqCell */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromSeqCell :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the arrival time (falling and rising)
     - Compute the transition(falling and rising)
   on the net "GnlAssocActualPort (Assoc)".
*/

```

timecomp2.c

```
/*
 *-----*
 GNL_STATUS GnlComputeArrivalTimeFromSeqCell (GNL_ASSOC          Assoc,
                                              float   *MaxTimeRise,
                                              float   *MaxTimeFall,
                                              float   *MaxOutTransRise,
                                              float   *MaxOutTransFall,
                                              LIBC_LIB    Lib,
                                              int      Tag,
                                              int      WithRec)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR        Var, Vari;
    GNL_TIMING_INFO TimingI;
    float         InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall;
    float         Length, WireCapa, WireResistance, OutCapa;
    BLIST         Interface;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_STATUS    GnlStatus;
    unsigned int   Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinOut) == INPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    Interface = GnlSequentialCompoInterface (SeqCompo);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if (!AssocI)
            continue;
        Vari = GnlAssocActualPort (AssocI);
        if (!Vari)
            continue;
```

timecomp2.c

```
if (GnlVarIsVar (VarI))
    if (GnlVarIsVdd (VarI) || GnlVarIsVss (VarI))
        continue;

Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if (GnlComputeArrivalTimeFromVar (VarI, Lib, Tag, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
    return (GNL_MEMORY_FULL);
InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
    Fanout, Cell, PinIn, PinOut,
    &DelayR, &OutTransR,
    &DelayF, &OutTransF);

if (*MaxOutTransRise < OutTransR)
    *MaxOutTransRise = OutTransR ;
if (*MaxOutTransFall < OutTransF)
    *MaxOutTransFall = OutTransF;

GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
    DelayR, DelayF,
    GnlTimingInfoArrivalRise (TimingI),
    GnlTimingInfoArrivalFall (TimingI),
    &TimeRise, &TimeFall);

if (*MaxTimeRise < TimeRise)
    *MaxTimeRise = TimeRise;
if (*MaxTimeFall < TimeFall)
    *MaxTimeFall = TimeFall;
}
return (GNL_OK);

}

/*-----*/
/* GnlComputeArrivalTimeFromUserCompo */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromUserCompo :
 - For a given GNL_ASSOC (Assoc) :
     - Compute the arrival time (falling and rising)
     - Compute the transition(falling and rising)
 on the net "GnlAssocActualPort (Assoc)"; this net is connected to
 hierachycal block or combinational cell
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromUserCompo (GNL_ASSOC      Assoc,
                                              float          *TimeRise,
```

timecomp2.c

```
        float      *TimeFall,
        float      *TransRise,
        float      *TransFall,
        LIBC_LIB Lib,
        GNL_VAR   Actual,
        int       Tag,
        int       WithRec)
{
    GNL_VAR          Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS        GnlStatus;
    int              Rank;
    unsigned int      Key;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);

Formal = GnlAssocFormalPort (Assoc);
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        if (GnlStatus = GnlComputeArrivalTimeAtOutputOfCombCell (UserCompo,
                                                               Assoc, TimeRise,
                                                               TimeFall, TransRise,
                                                               TransFall, Lib,
                                                               Tag, WithRec))
            return (GnlStatus);

        return (GNL_OK);
    }
    else
    /* It is a black box without any definition so we stop */
    /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

/* If the actual net is a VDD or a VSS then no timing to compute */
if (GnlVarIsVar (Actual))
    if (GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))
    {
        *TimeRise = 0.0;
        *TimeFall = 0.0;
        *TransRise = 0.0;
        *TransFall = 0.0;
        return (GNL_OK);
    }

if (BListAddElt (G_PileOfComponent, UserCompo))
```

timecomp2.c

```
    return (GNL_MEMORY_FULL);

    if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
                                                    &AuxFormal))
        return (GnlStatus);
    if (GnlComputeArrivalTimeFromVar (AuxFormal, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal, &TimingInfo,
&Key,
                                         &Rank))
        return (GNL_MEMORY_FULL);
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*TimeRise = GnlTimingInfoArrivalRise (TimingInfo);
*TimeFall = GnlTimingInfoArrivalFall (TimingInfo);
*TransRise = GnlTimingInfoTransitionRise (TimingInfo);
*TransFall = GnlTimingInfoTransitionFall (TimingInfo);

    return (GNL_OK);
}

/*-----
/* GnlComputeArrivalTimeFromAssoc
/*-----
/* Doc procedure GnlComputeArrivalTimeFromAssoc :
- For a given GNL_ASSOC (Assoc) :
- For a given GNL_ASSOC (Assoc) :
    - Compute the arrival time (falling and rising)
    - Compute the transition(falling and rising)
on the net "GnlAssocActualPort (Assoc)".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromAssoc (GNL_ASSOC Assoc,
                                             float      *TimeRise,
                                             float      *TimeFall,
                                             float      *TransRise,
                                             float      *TransFall,
                                             LIBC_LIB Lib,
                                             GNL_VAR   Actual,
                                             int        Tag,
                                             int        WithRec)
{
    GNL_STATUS          GnlStatus;
    int                 Rank;
    GNL_COMPONENT       GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            *TimeRise = *TimeFall = *TransRise = *TransFall = 0.0;
            break;
            if (GnlStatus = GnlComputeArrivalTimeFromSeqCell (Assoc,
TimeRise,
                                         TimeFall, TransRise, TransFall, Lib, Tag, WithRec))
                return (GnlStatus);
    }
}
```

timecomp2.c

```
        return (GnlStatus);
        break;

    case GNL_USER_COMPO:
        if (GnlStatus = GnlComputeArrivalTimeFromUserCompo (Assoc,
                                                       TimeRise, TimeFall,
                                                       TransRise, TransFall, Lib,
                                                       Actual, Tag, WithRec))
            return (GnlStatus);
        break;

    case GNL_TRISTATE_COMPO:
        if (GnlStatus = GnlComputeArrivalTimeFrom3State (Assoc,
                                                       TimeRise, TimeFall,
                                                       TransRise, TransFall, Lib, Tag,
                                                       WithRec))
            return (GnlStatus);
        break;

    case GNL_MACRO_COMPO:
        break;

    default:
        GnlError (12 /* Unknown component */);
    }

    return (GNL_OK);
}
/*-----*/
/* GnlComputeSetupHoldTimeFromSeqCompo */
/*-----*/
GNL_STATUS GnlComputeSetupHoldTimeFromSeqCompo (
    GNL_SEQUENTIAL_COMPONENT SeqCompo,
    char                  *PinInName,
    GNL_TIMING_INFO      TimingInfoOfPin,
    float                 *TimeRise,
    float                 *TimeFall,
    LIBC_LIB               Lib,
    int                  WithRec)
{
    LIBC_CELL           Cell;
    LIBC_PIN             PinClock, PinIn;
    int                  Fanout, Rank;
    GNL_ASSOC            AssocClock;
    GNL_VAR              VarClock;
    char                 *ClockName;
    GNL_TIMING_INFO     TimingClock;
    float                InTransRise, InTransFall, OutTransR, OutTransF;
    float                Length, WireCapa, OutCapa;
    BLIST                Interface;
    GNL_STATUS           GnlStatus;
    unsigned int          Key;

    *TimeRise = 0.0;
    *TimeFall = 0.0;
    Cell = GnlSeqCompoInfoCell (SeqCompo);
```

timecomp2.c

```
if (!(PinIn = LibGetPinFromNameAndCell (Cell, PinInName)))
    return (GNL_OK);
AssocClock = GnlSequentialCompoClockAssoc (SeqCompo);
ClockName = (char *) GnlAssocFormalPort (AssocClock);
VarClock = GnlAssocActualPort (AssocClock);
if (GnlVarIsVar (VarClock))
    if (GnlVarIsVdd (VarClock) || GnlVarIsVss (VarClock))
        return (GNL_OK);

if (!(PinClock = LibGetPinFromNameAndCell (Cell, ClockName)))
    return (GNL_OK);

if (GnlComputeArrivalTimeFromVar (VarClock, Lib, 0, WithRec, 1, 0))
    return (GNL_MEMORY_FULL);
if (GnlTimingInfoCorrespToCurrentPathFromVar (VarClock,
                                              &TimingClock, &Key, &Rank))
    return (GNL_MEMORY_FULL);
InTransRise = GnlTimingInfoTransitionRise (TimingClock);
InTransFall = GnlTimingInfoTransitionFall (TimingClock);

Fanout = GnlTimingInfoFanout (TimingInfoOfPin);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingInfoOfPin);

LibDelayArcCellRiseSetup (InTransRise, InTransFall, OutCapa, Cell, PinIn,
                           PinClock, TimeRise);
LibDelayArcCellFallSetup (InTransRise, InTransFall, OutCapa, Cell, PinIn,
                           PinClock, TimeFall);

return (GNL_OK);
}

/*-----*/
/* GnlComputeSetupHoldTimeFromVar */
/*-----*/
GNL_STATUS GnlComputeSetupHoldTimeFromVar (GNL_VAR      Var,
                                             GNL_TIMING_INFO  TimingInfoOfVar,
                                             LIBC_LIB Lib,
                                             float      *MaxSetupTimeR,
                                             float      *MaxSetupTimeF)
{
    GNL_ASSOC      AssocI;
    GNL_STATUS     GnlStatus;
    BLIST          AssocDests, LeftVarAssigneds;
    GNL_COMPONENT   GnlCompo;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    int             i;
    float           MaxTimeR, MaxTimeF, TimeRise, TimeFall;
    char            *Formal;

    if (GnlGetDestinationsOfVar (Var, &AssocDests, &LeftVarAssigneds))
        return (GNL_MEMORY_FULL);

    BListQuickDelete (&LeftVarAssigneds);

    *MaxSetupTimeR = *MaxSetupTimeF = 0.0;
```

timecomp2.c

```
for (i=0; i < BListSize (AssocDest); i++)
{
    AssocI = (GNL_ASSOC) BlistElt (AssocDest, i);
    GnlCompo = GnlAssocTraversalInfoComponent (AssocI);
    if (GnlComponentType (GnlCompo) != GNL_SEQUENTIAL_COMPO)
        continue;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlCompo;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (GnlStatus = GnlComputeSetupHoldTimeFromSeqCompo (SeqCompo, Formal,
        TimingInfoOfVar, &TimeRise, &TimeFall, Lib, 0))
        return (GnlStatus);
    if (*MaxSetupTimeR < TimeRise)
        *MaxSetupTimeR = TimeRise;
    if (*MaxSetupTimeF < TimeFall)
        *MaxSetupTimeF = TimeFall;
}
BListQuickDelete (&AssocDest);

return (GNL_OK);
}

/*-----*/
/* GnlComputeArrivalTimeFromVar */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromVar :
   - For a given GNL_VAR (Var) :
     - Compute the arrival time (falling and rising) of
       the net "Var".
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromVar (GNL_VAR      Var,
                                         LIBC_LIB     Lib,
                                         int          Tag,
                                         int          WithRec,
                                         int          InoutIsIn,
                                         int          VarIsInout)
{
    GNL_VAR          AuxVar, Actual;
    GNL_ASSOC        AuxAssocVar;
    int              i, Fanout, Rank;
    GNL_STATUS       GnlStatus;
    BLIST            AssocSources, RightVarAssigneds;
    GNL_TIMING_INFO TimingInfo, AuxTimingInfo;
    GNL_USER_COMPONENT InstOfGnl;
    float            TimeRise, TimeFall, TransRise, TransFall, MaxTime,
                    Time;
    float            MaxTimeRise, MaxTimeFall, MaxTransRise, MaxTransFall;
    float            MaxSetupTimeR, MaxSetupTimeF;
    unsigned int     Key;
    int              AuxVarIsInout;

    if (GnlVarIsVar (Var))
        if (GnlVarIsVdd (Var) || GnlVarIsVss (Var))
            return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
```

timecomp2.c

```
        &TimingInfo, &Key, &Rank)))  
    return (GnlStatus);  
  
if ((GnlVarDir (Var) == GNL_VAR_INOUT || VarIsInout) && InoutIsIn)  
    TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);  
  
if (GnlTimingInfoTag (TimingInfo) == Tag)  
    return (GNL_OK);  
SetGnlTimingInfoTag (TimingInfo, Tag);  
  
TimeRise = TimeFall = MaxTimeRise = MaxTimeFall = 0.0;  
TransRise = TransFall = MaxTransRise = MaxTransFall = 0.0;  
  
if (GnlVarDir (Var) == GNL_VAR_INPUT ||  
    (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn) )  
{  
    if (!BListSize (G_PileOfComponent))  
    {  
        /* Take the constraint from user (for later) */  
        /*if (GnlVarDir (Var) == GNL_VAR_INPUT ||  
         (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)) */  
        {  
            SetGnlTimingInfoArrivalRise (TimingInfo, MaxTimeRise);  
            SetGnlTimingInfoArrivalFall (TimingInfo, MaxTimeFall);  
            SetGnlTimingInfoTransitionRise (TimingInfo, MaxTransRise);  
            SetGnlTimingInfoTransitionFall (TimingInfo, MaxTransFall);  
  
            return (GNL_OK);  
        }  
    }  
    else  
    {  
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,  
                                         BListSize (G_PileOfComponent)-1);  
  
        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));  
        AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,  
                                                          &Actual);  
        if (!AuxAssocVar)  
        {  
            if (BListAddElt (G_PileOfComponent, InstOfGnl))  
                return (GNL_MEMORY_FULL);  
            return (GNL_OK);  
        }  
        if (!(GnlVarIsVar (Actual) &&  
              (GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))))  
        {  
            AuxVarIsInout = (GnlVarDir (Var) == GNL_VAR_INOUT);  
            if (GnlComputeArrivalTimeFromVar (Actual, Lib, Tag,  
                                              WithRec, 1, AuxVarIsInout))  
                return (GNL_MEMORY_FULL);  
            if (GnlTimingInfoCorrespToCurrentPathFromVar  
(Actual,&AuxTimingInfo,  
                                         &Key, &Rank))  
                return (GNL_MEMORY_FULL);  
            if (GnlVarDir (Actual) == GNL_VAR_INOUT)
```

timecomp2.c

```
TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook
(AuxTimingInfo);
    MaxTimeRise = GnlTimingInfoArrivalRise (AuxTimingInfo);
    MaxTimeFall = GnlTimingInfoArrivalFall (AuxTimingInfo);
    MaxTransRise = GnlTimingInfoTransitionRise (AuxTimingInfo);
    MaxTransFall = GnlTimingInfoTransitionFall (AuxTimingInfo);
}
if (BListAddElt (G_PileOfComponent, InstOfGnl))
    return (GNL_MEMORY_FULL);
}

if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (RightVarAssigneds, i);

    if (GnlVarIsVar (AuxVar))
        if (GnlVarIsVdd (AuxVar) || GnlVarIsVss (AuxVar))
            continue;
    if (GnlVarIsVar (AuxVar))
    if (GnlVarIsVdd (AuxVar) || GnlVarIsVss (AuxVar))
        continue;

    if (GnlComputeArrivalTimeFromVar (AuxVar, Lib, Tag, WithRec, 1, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar, &AuxTimingInfo,
                                                &Key, &Rank))
        return (GNL_MEMORY_FULL);

    MaxTime = MaxTimeRise;
    if (MaxTime < MaxTimeFall)
        MaxTime = MaxTimeFall;
    Time = GnlTimingInfoArrivalRise (AuxTimingInfo);
    if (Time < GnlTimingInfoArrivalFall (AuxTimingInfo))
        Time = GnlTimingInfoArrivalFall (AuxTimingInfo);

    if (MaxTime < Time)
    {
        MaxTimeRise = GnlTimingInfoArrivalRise (AuxTimingInfo);
        MaxTimeFall = GnlTimingInfoArrivalFall (AuxTimingInfo);
        MaxTransRise = GnlTimingInfoTransitionRise (AuxTimingInfo);
        MaxTransFall = GnlTimingInfoTransitionFall (AuxTimingInfo);
    }
}
BListQuickDelete (&RightVarAssigneds);

for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    TimeRise = TimeFall = 0.0;
    TransRise = TransFall = 0.0;
    if ((GnlStatus = GnlComputeArrivalTimeFromAssoc (AuxAssocVar, &TimeRise,
                                                    &TimeFall, &TransRise,
```

timecomp2.c

```
&TransFall, Lib, Var,
Tag, WithRec)))
return (GnlStatus);

MaxTime = MaxTimeRise;
if (MaxTime < MaxTimeFall)
    MaxTime = MaxTimeFall;
Time = TimeRise;
if (Time < TimeFall)
    Time = TimeFall;

if (MaxTime < Time)
{
    MaxTimeRise = TimeRise;
    MaxTimeFall = TimeFall;
    MaxTransRise = TransRise;
    MaxTransFall = TransFall;
}
BListQuickDelete (&AssocSources);

SetGnlTimingInfoArrivalRise (TimingInfo, MaxTimeRise);
SetGnlTimingInfoArrivalFall (TimingInfo, MaxTimeFall);
SetGnlTimingInfoTransitionRise (TimingInfo, MaxTransRise);
SetGnlTimingInfoTransitionFall (TimingInfo, MaxTransFall);

/* *MaxSetupTimeR = *MaxSetupTimeF = 0.0;
GnlComputeSetupHoldTimeFromVar (Var, TimingInfo, Lib,
                                &MaxSetupTimeR, &MaxSetupTimeF);
SetGnlTimingInfoArrivalRise (TimingInfo,
                             GnlTimingInfoArrivalRise (TimingInfo) + MaxSetupTimeR);
SetGnlTimingInfoArrivalFall (TimingInfo,
                            GnlTimingInfoArrivalFall (TimingInfo) + MaxSetupTimeF);

if (MaxSetupTimeR || MaxSetupTimeF)
    if (GnlStatus = GnlUpdateArrTimeOfRightVarAssigneds (Var))
        return (GnlStatus); */

return (GNL_OK);
}

/*
* TimePutAssocSourceOfVarOnRight
*/
static void TimePutAssocSourceOfVarOnRight (GNL_VAR Var)
{
    int          i, j, el;
    GNL_ASSOC AssocI;
    BLIST        AssocSources, RightVarAssigneds, ListAssoc;

    GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds);
    ListAssoc = GnlVarTraversalInfoListAssoc (Var);
    for (i=0; i < BListSize (AssocSources); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (AssocSources, i);
        j = BListMemberOfList (ListAssoc, AssocI, IntIdentical);
```

timecomp2.c

```
el = BListElt (ListAssoc, BListSize (ListAssoc) -1);
BListElt (ListAssoc, BListSize (ListAssoc) -1) = BListElt (ListAssoc, j -1);
    BListElt (ListAssoc, j -1) = el;
}
BListQuickDelete (&AssocSources);
BListQuickDelete (&RightVarAssigneds);
}
/*-----*/
/* GnlComputeArrivalTimeFromGnl */
/*-----*/
/* Doc procedure GnlComputeArrivalTimeFromGnl :
   - For a given GNL (Gnl) :
      - Hierarchically compute the arrival time (falling and rising) on each
        net GNL_VAR.
*/
/*-----*/
GNL_STATUS GnlComputeArrivalTimeFromGnl (GNL Gnl,
                                           LIBC_LIB Lib,
                                           int Tag)
{
    GNL_VAR Var;
    int i, j;
    GNL_STATUS GnlStatus;
    GNL GnlCompoI;
    GNL_COMPONENT ComponentI;
    BLIST Components, BucketI;

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT)BListElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
            continue;
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlComputeArrivalTimeFromGnl (GnlCompoI, Lib, Tag))
                return (GnlStatus);
            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        }
    }

    for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
    {
        BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
        for (j=0; j < BListSize (BucketI); j++)
        {
            Var = (GNL_VAR)BListElt (BucketI, j);

            if (!Var)
                continue;

            if (GnlVarIsVar (Var))
```

timecomp2.c

```

        if (GnlVarIsVdd (Var) || GnlVarIsVss (Var))
            continue;

        TimePutAssocSourceOfVarOnRight (Var);
        if (GnlComputeArrivalTimeFromVar (Var, Lib, Tag, 1, 0, 0))
            return (GNL_MEMORY_FULL);
    }

}

return (GNL_OK);
}

/*
 * GnlComputeArrivalTimeFromNetwork
 */
/*
 * Doc procedure GnlComputeArrivalTimeFromNetwork :
 * - For a given Network (GNL_NETWORK) :
 *   - Hierarchically compute the arrival time (falling and rising) on each
 *     net GNL_VAR.
 */

GNL_STATUS GnlComputeArrivalTimeFromNetwork (GNL_NETWORK      Nw,
                                              LIBC_LIB       Lib)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    int          Tag;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
    Tag = GnlNetworkTag (Nw);
    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlComputeArrivalTimeFromGnl (TopGnl, Lib, Tag)))
        return (GnlStatus);

    BListQuickDelete (&G_PileOfComponent);
    return (GNL_OK);
}

/*
 * -----
 * -----
 */

/*
 * -----
 * -----
 * GnlGetRequiredTimeRiseAndFallSwitchTimingSence (LIBC_PIN PinIn, LIBC_PIN
 * PinOut,
 *                                                 float DelayR, float DelayF,
 *                                                 float TimeRiseOut, float TimeFallOut,
 */

```

timecomp2.c

```
float *TimeRise, float *TimeFall)
{
    LIBC_TIMING          Timing;
    *TimeRise = *TimeFall = MAX_FLOAT;
    if (!DelayR && !DelayF)
        return;
    Timing = LibGetArcTiming (PinIn, PinOut);
    if (!Timing)
        return;
    switch (LibTimingSense (Timing))
    {
        case POSITIVE_UNATE_E:
            *TimeRise = TimeRiseOut - DelayR;
            *TimeFall = TimeFallOut - DelayF;
            break;
        case NEGATIVE_UNATE_E:
            *TimeFall = TimeRiseOut - DelayR;
            *TimeRise = TimeFallOut - DelayF;
            break;
        case NON_UNATE_E:
            *TimeRise = TimeRiseOut - DelayR;
            if (*TimeRise > TimeFallOut - DelayF)
                *TimeRise = TimeFallOut - DelayF;
            *TimeFall = TimeFallOut - DelayF;
            if (*TimeFall > TimeRiseOut - DelayR)
                *TimeFall = TimeRiseOut - DelayR;
            break;
    }
}
/*-----*/
/* GnlComputeRequiredTimeAtOutputOfCombCell */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeAtOutputOfCombCell :
   - For a given a GNL_USER_COMPONENT (UserCompo) generic cell and
     GNL_ASSOC (AssocOutput) (output GNL_ASSOC of UserCompo) :
       - Compute the required time (falling and rising)
*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeAtOutputOfCombCell(
    GNL_USER_COMPONENT UserCompo,
    GNL_ASSOC AssocInput,
    float *MinTimeRise,
    float *MinTimeFall,
    LIBC_LIB      Lib,
    int           Tag,
    int           WithRec)
{
    LIBC_CELL          Cell;
```

timecomp2.c

```
LIBC_PIN      PinOut, PinIn;
int          i, Fanout, Rank;
GNL_ASSOC    AssocI;
char         *Formal;
GNL_VAR      Var, VarI;
GNL_TIMING_INFO TimingI;
float        InTransRise, InTransFall, OutTransR, OutTransF,
             DelayR, DelayF, TimeRise, TimeFall, MinTime, Time;
float        Length, WireCapa, WireResistance, OutCapa;
BLIST        Interface;
GNL_STATUS   GnlStatus;
unsigned int Key;

Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
Var = GnlAssocActualPort (AssocInput);
Formal = (char *) GnlAssocFormalPort (AssocInput);

if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E )
    return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                           &TimingI, &Key, &Rank)))
    return (GnlStatus);

if (GnlVarDir (Var) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

Interface = GnlUserComponentInterface (UserCompo);
for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinOut) != OUTPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if (WithRec)
        if (GnlComputeRequiredTimeFromVar (VarI, Lib, Tag, WithRec, 0, 0))
            return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
```

timecomp2.c

```
Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);

GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoRequiredRise (TimingI),
                                                GnlTimingInfoRequiredFall (TimingI),
                                                &TimeRise, &TimeFall);

MinTime = *MinTimeRise;
if (MinTime > *MinTimeFall)
    MinTime = *MinTimeFall;
Time = TimeRise;
if (Time > TimeFall)
    Time = TimeFall;

if (MinTime > Time)
{
    *MinTimeRise = TimeRise;
    *MinTimeFall = TimeFall;
}
}
return (GNL_OK);
}

/*-----
/* GnlComputeRequiredTimeFromUserCompo
*/
/*-----
/* Doc procedure GnlComputeRequiredTimeFromUserCompo :
   - For a given GNL_ASSOC (Assoc) :
      - Compute the required time (falling and rising)
        on the net "GnlAssocActualPort (Assoc)".
   - This GNL_ASSOC is connected to a user component.
*/
/*-----
GNL_STATUS GnlComputeRequiredTimeFromUserCompo (GNL_ASSOC Assoc,
                                                 float      *TimeRise,
                                                 float      *TimeFall,
                                                 LIBC_LIB Lib,
                                                 GNL_VAR   Actual,
                                                 int       Tag,
                                                 int       WithRec)
{
    GNL_VAR          Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS       GnlStatus;
    int             Rank;
    unsigned int     Key;
    int             AuxVarIsInout;
```

timecomp2.c

```
UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Var = GnlAssocActualPort (Assoc);

Formal = GnlAssocFormalPort (Assoc);
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        if (GnlStatus = GnlComputeRequiredTimeAtOutputOfCombCell (UserCompo,
                                                               Assoc, TimeRise,
                                                               TimeFall, Lib, Tag, WithRec))
            return (GnlStatus);

        return (GNL_OK);
    }
    else
    /* It is a black box without any definition so we stop */
    /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
    return (GnlStatus);

AuxVarIsInout = (GnlVarDir (Actual) == GNL_VAR_INOUT);
if (GnlComputeRequiredTimeFromVar (AuxFormal, Lib, Tag, WithRec, 1,
AuxVarIsInout))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal,
                                                               &TimingInfo, &Key, &Rank)))
    return (GnlStatus);
if (GnlVarDir (AuxFormal) == GNL_VAR_INOUT)
    TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

*TimeRise = GnlTimingInfoRequiredRise (TimingInfo);
*TimeFall = GnlTimingInfoRequiredFall (TimingInfo);

return (GNL_OK);
}
/*-----*/
/* GnlComputeRequiredTimeFromSeqCell */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromSeqCell :
 - For a given GNL_ASSOC (Assoc) :
```

timecomp2.c

```
- Compute the required time (falling and rising)
on the net "GnlAssocActualPort (Assoc)".
- This GNL_ASSOC is connected to a sequential component.

*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromSeqCell (GNL_ASSOC      Assoc,
                                              float        *MinTimeRise,
                                              float        *MinTimeFall,
                                              LIBC_LIB    Lib,
                                              int         Tag,
                                              int         WithRec)

{
    GNL_SEQUENTIAL_COMPONENT      SeqCompo;
    LIBC_CELL        Cell;
    LIBC_PIN         PinOut, PinIn;
    int              i, Fanout, Rank;
    GNL_ASSOC        AssocI;
    char             *Formal;
    GNL_VAR          Var, VarI;
    GNL_TIMING_INFO TimingI;
    float            InTransRise, InTransFall, OutTransR, OutTransF,
                    DelayR, DelayF, TimeRise, TimeFall, Time, MinTime;
    float            Length, WireCapa, WireResistance, OutCapa;
    BLIST            Interface;
    GNL_STATUS       GnlStatus;
    unsigned int     Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E )
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    Interface = GnlSequentialCompoInterface (SeqCompo);
    for (i=0; i < BlistSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC)BListElt (Interface, i);
        if (!AssocI)
            continue;
        VarI = GnlAssocActualPort (AssocI);
        if (!VarI)
            continue;
        if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
            continue;
```

timecomp2.c

```
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinOut) != OUTPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if (GnlComputeRequiredTimeFromVar (VarI, Lib, Tag, WithRec, 0, 0))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);

Fanout = GnlTimingInfoFanout (TimingI);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);

GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoRequiredRise (TimingI),
                                                GnlTimingInfoRequiredFall (TimingI),
                                                &TimeRise, &TimeFall);

MinTime = *MinTimeRise;
if (MinTime > *MinTimeFall)
    MinTime = *MinTimeFall;
Time = TimeRise;
if (Time > TimeFall)
    Time = TimeFall;

if (MinTime > Time)
{
    *MinTimeRise = TimeRise;
    *MinTimeFall = TimeFall;
}
}
return (GNL_OK);
}
/*-----*/
/* GnlComputeRequiredTimeFrom3State */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFrom3State :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the required time (falling and rising)
       on the net "GnlAssocActualPort (Assoc)".
     - This GNL_ASSOC is connected to a tristate component.
*/
/*-----*/
```

timecomp2.c

```
GNL_STATUS GnlComputeRequiredTimeFrom3State (GNL_ASSOC          Assoc,
                                              float      *MinTimeRise,
                                              float      *MinTimeFall,
                                              LIBC_LIB  Lib,
                                              int       Tag,
                                              int       WithRec)

{
    GNL_TRISTATE_COMPONENT      TriState;
    LIBC_CELL       Cell;
    LIBC_PIN        PinOut, PinIn;
    int             i, Fanout, Rank;
    GNL_ASSOC       AssocI;
    char            *Formal;
    GNL_VAR         Var, Vari;
    GNL_TIMING_INFO TimingI;
    float           InTransRise, InTransFall, OutTransR, OutTransF,
                    DelayR, DelayF, TimeRise, TimeFall, MinTime, Time;
    float           Length, WireCapa, WireResistance, OutCapa;
    BLIST           Interface;
    GNL_STATUS      GnlStatus;
    unsigned int    Key;

    TriState = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Cell = GnlTriStateCompoInfoCell (TriState);
    Var = GnlAssocActualPort (Assoc);
    Formal = (char *) GnlAssocFormalPort (Assoc);

    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        return (GNL_OK);
    if (LibPinDirection (PinIn) == OUTPUT_E)
        return (GNL_OK);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    if (GnlVarDir (Var) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    Interface = GnlTriStateInterface (TriState);
    for (i=0; i < BListSize (Interface); i++)
    {
        AssocI = (GNL_ASSOC) BListElt (Interface, i);
        if (!AssocI)
            continue;
        Vari = GnlAssocActualPort (AssocI);
        if (!Vari)
            continue;
        if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
            continue;
        Formal = (char *) GnlAssocFormalPort (AssocI);
        if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
            continue;
        if (LibPinDirection(PinOut) != OUTPUT_E)
```

timecomp2.c

```
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if (WithRec)
    if (GnlComputeRequiredTimeFromVar (VarI, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                    Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Cell, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);

    GnlGetRequiredTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                    DelayR, DelayF,
                                                    GnlTimingInfoRequiredRise (TimingI),
                                                    GnlTimingInfoRequiredFall (TimingI),
                                                    &TimeRise, &TimeFall);

    MinTime = *MinTimeRise;
    if (MinTime > *MinTimeFall)
        MinTime = *MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    if (MinTime > Time)
    {
        *MinTimeRise = TimeRise;
        *MinTimeFall = TimeFall;
    }
}
return (GNL_OK);
}
/*-----*/
/* GnlComputeRequiredTimeFromAssoc */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromAssoc :
   - For a given GNL_ASSOC (Assoc) :
     - Compute the required time (falling and rising)
       on the net "GnlAssocActualPort (Assoc)".
*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromAssoc (GNL_ASSOC Assoc,
                                             float      *TimeRise,
                                             float      *TimeFall,
                                             LIBC_LIB Lib,
```

timecomp2.c

```
        GNL_VAR Actual,
        int      Tag,
        int      WithRec)
{
    GNL_STATUS          GnlStatus;
    GNL_COMPONENT       GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            *TimeRise = *TimeFall = 0.0;
            break;
        if (GnlStatus = GnlComputeRequiredTimeFromSeqCell (Assoc,
TimeRise,
                                         TimeFall, Lib, Tag, WithRec))
            return (GnlStatus);
        break;

        case GNL_USER_COMPO:
            if (GnlStatus = GnlComputeRequiredTimeFromUserCompo (Assoc,
                                         TimeRise, TimeFall, Lib,
                                         Actual, Tag, WithRec))
                return (GnlStatus);
            break;

        case GNL_TRISTATE_COMPO:
            if (GnlStatus = GnlComputeRequiredTimeFrom3State (Assoc,
                                         TimeRise, TimeFall, Lib,
                                         Tag, WithRec))
                return (GnlStatus);
            break;

        case GNL_MACRO_COMPO:
            break;

        default:
            GnlError (12 /* Unknown component */);
            break;
    }

    return (GNL_OK);
}
/*-----*/
/* TimeSortListByRequiredTime */
/*-----*/
static void TimeSortListByRequiredTime (BLIST List, BLIST ListRequireTime)
{
    int      i, k, IndexMin, el;
    float    Min;

    k = 0;
    while (k < BListSize (ListRequireTime))
    {
        Min = MAX_FLOAT;
        IndexMin = k;
```

timecomp2.c

```
for (i = k; i < BListSize (ListRequireTime); i++)
{
    if (*((float *) BListElt (ListRequireTime, i)) < Min)
    {
        IndexMin = i;
        Min = *((float *) BListElt (ListRequireTime, i));
    }
}
el = BListElt (ListRequireTime, IndexMin);
BListElt (ListRequireTime, IndexMin) = BListElt (ListRequireTime, k);
BListElt (ListRequireTime, k) = el;

el = BListElt (List, IndexMin);
BListElt (List, IndexMin) = BListElt (List, k);
BListElt (List, k) = el;
k++;
}

}

/*-----*/
/* GnlComputeRequiredTimeFromVar */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromVar :
   - For a given GNL_VAR (Var) :
     - Compute the required time (falling and rising) of
       the net "Var".
*/
/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromVar (GNL_VAR Var,
                                             LIBC_LIB Lib,
                                             int Tag,
                                             int WithRec,
                                             int InoutIsIn,
                                             int VarIsInout)
{
    GNL_VAR AuxVar, Actual;
    GNL_ASSOC AuxAssocVar;
    int i, Fanout, Rank, Elt;
    GNL_STATUS GnlStatus;
    BLIST AssocDest, LeftVarAssigneds,
ListAssoc, ListRequireTime;
    GNL_TIMING_INFO TimingInfo, AuxTimingInfo;
    GNL_USER_COMPONENT InstOfGnl;
    float TimeRise, TimeFall, Time, MinTime, *Real;
    float MinTimeRise, MinTimeFall;
    float MaxSetupTimeR, MaxSetupTimeF;
    unsigned int Key;
    int AuxVarIsInout;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    if ((GnlVarDir (Var) == GNL_VAR_INOUT || VarIsInout) && InoutIsIn)
        TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);
```

timecomp2.c

```
if (GnlTimingInfoTag (TimingInfo) == Tag)
    return (GNL_OK);
SetGnlTimingInfoTag (TimingInfo, Tag);
TimeRise = TimeFall = MinTimeRise = MinTimeFall = 0.0;

if ( (GnlVarDir (Var) == GNL_VAR_OUTPUT) ||
     (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn) )
{
    if (!BListSize (G_PileOfComponent))
    {
        /* Take the constraint from user (for later) */
        /*if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
            (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn))*/
    {
        SetGnlTimingInfoRequiredRise (TimingInfo, MinTimeRise);
        SetGnlTimingInfoRequiredFall (TimingInfo, MinTimeFall);
        return (GNL_OK);
    }
}
else
{
    InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                          &Actual);
    if (!AuxAssocVar)
    {
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }
    if (GnlComputeRequiredTimeFromVar (Actual, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Actual,
                                                                &AuxTimingInfo, &Key, &Rank)))
        return (GnlStatus);

    TimeRise = GnlTimingInfoRequiredRise (AuxTimingInfo);
    TimeFall = GnlTimingInfoRequiredFall (AuxTimingInfo);
    if (BListAddElt (G_PileOfComponent, InstOfGnl))
        return (GNL_MEMORY_FULL);

    MinTime = MinTimeRise;
    if (MinTime > MinTimeFall)
        MinTime = MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    if (MinTime > Time)
    {
        MinTimeRise = TimeRise;
        MinTimeFall = TimeFall;
    }
}
```

timecomp2.c

```
}

if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

ListAssoc = GnlVarTraversalInfoListLeftAssign (Var);
if (BListCreateWithSize (BListSize (LeftVarAssigneds), &ListRequireTime))
    return (GNL_MEMORY_FULL);
for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (LeftVarAssigneds, i);
    if (GnlComputeRequiredTimeFromVar (AuxVar, Lib, Tag, WithRec, 0, 0))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (AuxVar,
                                                               &AuxTimingInfo, &Key, &Rank)))
        return (GnlStatus);

    MinTime = MinTimeRise;
    if (MinTime > MinTimeFall)
        MinTime = MinTimeFall;
    Time = GnlTimingInfoRequiredRise (AuxTimingInfo);
    if (Time > GnlTimingInfoRequiredFall (AuxTimingInfo))
        Time = GnlTimingInfoRequiredFall (AuxTimingInfo);

    Real = (float *) calloc (1, sizeof (float));
    (*Real) = Time;
    if (BListAddElt (ListRequireTime, (int) Real))
        return (GNL_MEMORY_FULL);

    if (MinTime > Time)
    {
        MinTimeRise = GnlTimingInfoRequiredRise (AuxTimingInfo);
        MinTimeFall = GnlTimingInfoRequiredFall (AuxTimingInfo);
    }
}
/* Tri of ListAssoc by required time */
TimeSortListByRequiredTime (ListAssoc, ListRequireTime);
BListDelete (&ListRequireTime, BListElemFree);
BListQuickDelete (&LeftVarAssigneds);
ListAssoc = GnlVarTraversalInfoListAssoc (Var);
if (BListCreateWithSize (BListSize (AssocDest), &ListRequireTime))
    return (GNL_MEMORY_FULL);
for (i=0; i < BListSize (AssocDest); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocDest, i);
    TimeRise = TimeFall = 0.0;
    if (GnlComputeRequiredTimeFromAssoc (AuxAssocVar, &TimeRise,
                                         &TimeFall, Lib, Var, Tag, WithRec))
        return (GNL_MEMORY_FULL);

    MinTime = MinTimeRise;
    if (MinTime > MinTimeFall)
        MinTime = MinTimeFall;
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;
```

timecomp2.c

```
Real = (float *) calloc (1, sizeof (float));
(*Real) = Time;
if (BListAddElt (ListRequireTime, (int) Real))
    return (GNL_MEMORY_FULL);

if (MinTime > Time)
{
    MinTimeRise = TimeRise;
    MinTimeFall = TimeFall;
}
/*
 * Tri of ListAssoc by required time */
TimeSortListByRequiredTime (ListAssoc, ListRequireTime);
BListDelete (&ListRequireTime, BListElemFree);
BListQuickDelete (&AssocDests);

SetGnlTimingInfoRequiredRise (TimingInfo, MinTimeRise);
SetGnlTimingInfoRequiredFall (TimingInfo, MinTimeFall);

/* *MaxSetupTimeR = *MaxSetupTimeF = 0.0;
GnlComputeSetupHoldTimeFromVar (Var, TimingInfo, Lib,
                                &MaxSetupTimeR, &MaxSetupTimeF);
SetGnlTimingInfoRequiredRise (TimingInfo,
    GnlTimingInfoRequiredRise (TimingInfo) - MaxSetupTimeR);
SetGnlTimingInfoRequiredFall (TimingInfo,
    GnlTimingInfoRequiredFall (TimingInfo) - MaxSetupTimeF);
*/
return (GNL_OK);
}

/*
 *-----*
/* GnlComputeRequiredTimeFromGnl                         */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromGnl :
   - For a given GNL (Gnl) :
      - Hierarchically compute the required time (falling and rising) on each
        net GNL_VAR.
*/
/*
 *-----*/
GNL_STATUS GnlComputeRequiredTimeFromGnl (GNL           Gnl,
                                           LIBC_LIB     Lib,
                                           int          Tag)
{
    GNL_VAR           Var;
    int              i, j;
    GNL_STATUS       GnlStatus;
    GNL             GnlCompoI;
    GNL_COMPONENT    ComponentI;
    BLIST            Components, BucketI;

    Components = GnlComponents (Gnl);
    for (i=0; i<BListSize (Components); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (Components, i);
        if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
```

timecomp2.c

```
    continue;
GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
if (GnlCompoI)
{
    if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
        return (GNL_MEMORY_FULL);
    if (GnlStatus = GnlComputeRequiredTimeFromGnl (GnlCompoI, Lib, Tag))
        return (GnlStatus);
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
}
}

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BlistElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BlistElt (BucketI, j);
        if (!Var || GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;
        if (GnlComputeRequiredTimeFromVar (Var, Lib, Tag, 1, 1, 0))
            return (GNL_MEMORY_FULL);
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlComputeRequiredTimeFromNetwork */
/*-----*/
/* Doc procedure GnlComputeRequiredTimeFromNetwork :
   - For a given Network (GNL_NETWORK) :
     - Hierarchically compute the required time (falling and rising) on each
       net GNL_VAR.
   */

/*-----*/
GNL_STATUS GnlComputeRequiredTimeFromNetwork (GNL_NETWORK Nw,
                                              LIBC_LIB Lib)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    int          Tag;

    SetGnlNetworkTag (Nw, GnlNetworkTag (Nw)+1);
    Tag = GnlNetworkTag (Nw);

    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlComputeRequiredTimeFromGnl (TopGnl, Lib, Tag)))
        return (GnlStatus);

    BListQuickDelete (&G_PileOfComponent);
    return (GNL_OK);
}
```

timecomp2.c

```
}

-----*/  
GnlGetSensFromArcTimingAndInputArrivalTime (LIBC_PIN PinIn, LIBC_PIN PinOut,  
                                         float TimeRiseIn, float TimeFallIn,  
                                         int *SensRise, int *SensFall)  
  
{  
    LIBC_TIMING          Timing;  
  
    Timing = LibGetArcTiming (PinIn, PinOut);  
  
    switch (LibTimingSense (Timing))  
    {  
        case POSITIVE_UNATE_E:  
            *SensRise = RISE;  
            *SensFall = FALL;  
            break;  
  
        case NEGATIVE_UNATE_E:  
            *SensRise = FALL;  
            *SensFall = RISE;  
            break;  
  
        case NON_UNATE_E:  
            *SensRise = RISE;  
            if (TimeRiseIn < TimeFallIn)  
                *SensRise = FALL;  
  
            *SensFall = FALL;  
            if (TimeFallIn < TimeRiseIn)  
                *SensFall = RISE;  
            break;  
    }  
}  
-----*/  
  
extern GNL_STATUS GnlGetCritPathFromVar (GNL_VAR      Var,  
                                         LIBC_LIB      Lib,  
                                         GNL_CRITICAL_PATH *CritPath,  
                                         GNL_CRITICAL_PATH CritSuccessorPath,  
                                         int   KindOfTime,  
                                         int *IsCombPath, int *IsSeqPath, int Start,  
                                         int VarIsInout);  
-----*/  
/* GnlGetCritPathAtOutputOfCombCell */  
-----*/  
GNL_STATUS GnlGetCritPathAtOutputOfCombCell (GNL_USER_COMPONENT UserCompo,  
                                              GNL_ASSOC      Assoc,  
                                              LIBC_LIB      Lib,  
                                              GNL_CRITICAL_PATH *CritPath,  
                                              GNL_CRITICAL_PATH CritSuccessorPath,  
                                              int *IsCombPath, int *IsSeqPath)  
{  
    LIBC_CELL          Cell;  
    LIBC_PIN          PinOut, PinIn;
```

timecomp2.c

```
int i, Fanout, Rank, SensRise, SensFall;
GNL_ASSOC AssocI;
char *Formal;
GNL_VAR Var, Vari;
GNL_TIMING_INFO TimingI;
float InTransRise, InTransFall, OutTransR, OutTransF,
      DelayR, DelayF, TimeRise, TimeFall;
float Length, WireCapa, WireResistance, OutCapa;
float SlackRise, SlackFall, MinSlack;
BLIST Interface;
GNL_STATUS GnlStatus;
int KindOfTime;
GNL_CRITICAL_PATH CritPredecessorPath;
unsigned int Key;

Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinOut) == INPUT_E)
    return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                          &TimingI, &Key, &Rank)))
    return (GnlStatus);

Fanout = GnlTimingInfoFanout (TimingI);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

Interface = GnlUserComponentInterface (UserCompo);
MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
            GnlCritPathArrTime (CritSuccessorPath);
for (i=0; i < BlistSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinIn) != INPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                              &TimingI, &Key, &Rank)))
        return (GnlStatus);
    if (GnlVarDir (VarI) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
```

timecomp2.c

```
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);

GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                               DelayR, DelayF,
                                               GnlTimingInfoArrivalRise (TimingI),
                                               GnlTimingInfoArrivalFall (TimingI),
                                               &TimeRise, &TimeFall);

if (!GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeRise,
4)
&& !GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeFall,
4))
    continue;

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathFanout (*CritPath, Fanout);
SetGnlCritPathArrTime (*CritPath,
                      GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath,
                      GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath,
                          GnlCritPathKindOfTime (CritSuccessorPath));
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                      GnlUserComponentInstName (UserCompo));
GnlGetSensFromArcTimingAndInputArrivalTime (PinIn, PinOut,
                                             GnlTimingInfoArrivalRise (TimingI),
                                             GnlTimingInfoArrivalFall (TimingI),
                                             &SensRise,&SensFall);
if (GnlCritPathKindOfTime (CritSuccessorPath) == RISE)
{
    SetGnlCritPathIncr (*CritPath, DelayR);
    KindOfTime = SensRise;
}
else
{
    SetGnlCritPathIncr (*CritPath, DelayF);
    KindOfTime = SensFall;
}

if (GnlGetCritPathFromVar (VarI, Lib,&CritPredecessorPath,
                           *CritPath, KindOfTime,
                           IsCombPath, IsSeqPath, 1, 0))
    return (GNL_MEMORY_FULL);
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
}
return (GNL_OK);
}/*-----*/
```

timecomp2.c

```
/* GnlGetCritPathFromUserCompo */  
/*-----*/  
GNL_STATUS GnlGetCritPathFromUserCompo (GNL_ASSOC Assoc,  
                                         LIBC_LIB Lib,  
                                         GNL_CRITICAL_PATH *CritPath,  
                                         GNL_CRITICAL_PATH CritSuccessorPath,  
                                         int KindOfTime,  
                                         int *IsCombPath, int *IsSeqPath)  
{  
    GNL_VAR Var, Formal;  
    GNL_USER_COMPONENT UserCompo;  
    GNL_TIMING_INFO TimingInfo;  
    GNL_STATUS GnlStatus;  
    int Rank;  
    GNL_CRITICAL_PATH CritPredecessorPath;  
    float SlackRise, SlackFall, MinSlack;  
    unsigned int Key;  
  
    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);  
    Var = GnlAssocActualPort (Assoc);  
  
    Formal = GnlAssocFormalPort (Assoc);  
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)  
    {  
        /* It corresponds to a LIBC library cell. */  
        if (GnlUserComponentCellDef (UserCompo))  
        {  
            if (GnlStatus = GnlGetCritPathAtOutputOfCombCell (UserCompo,  
                                                       Assoc, Lib, CritPath,  
                                                       CritSuccessorPath,  
                                                       IsCombPath, IsSeqPath))  
                return (GnlStatus);  
  
            return (GNL_OK);  
        }  
        else  
        /* It is a black box without any definition so we stop */  
        /* at its boundary. */  
        {  
            fprintf (stderr,  
                     " WARNING: black box <%s %s> may modify final estimation\n",  
                     GnlUserComponentName (UserCompo),  
                     GnlUserComponentInstName (UserCompo));  
        }  
        return (GNL_OK);  
    }  
  
    if (BListAddElt (G_PileOfComponent, UserCompo))  
        return (GNL_MEMORY_FULL);  
  
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Formal,  
                                                               &TimingInfo, &Key, &Rank)))  
        return (GnlStatus);  
  
    SlackRise = GnlTimingInfoRequiredRise (TimingInfo) -  
               GnlTimingInfoArrivalRise (TimingInfo);
```

timecomp2.c

```
SlackFall = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
            GnlCritPathArrTime (CritSuccessorPath);
if ( !GnlFloatEqual (MinSlack, SlackRise, 4)
    && !GnlFloatEqual (MinSlack, SlackFall, 4) )
{
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    return (GNL_OK);
}

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathArrTime (*CritPath, GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath, GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath, KindOfTime);
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathIncr (*CritPath, 0.0);

if (GnlGetCritPathFromVar (Formal, Lib, &CritPredecessorPath,
                           *CritPath, KindOfTime, IsCombPath, IsSeqPath, 0, 0))
    return (GNL_MEMORY_FULL);
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

return (GNL_OK);
}

/*-----*/
/* GnlGetCritPathFromSeqCell                               */
/*-----*/
GNL_STATUS GnlGetCritPathFromSeqCell (GNL_ASSOC Assoc,
                                       LIBC_LIB     Lib,
                                       GNL_CRITICAL_PATH *CritPath,
                                       GNL_CRITICAL_PATH CritSuccessorPath)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut;
    int           i, Rank;
    GNL_ASSOC     AssocI;
    char          *Formal;
    GNL_VAR        Var;
    GNL_TIMING_INFO TimingI;
    float          SlackRise, SlackFall, MinSlack;
    GNL_STATUS    GnlStatus;
    int           KindOfTime;
    GNL_CRITICAL_PATH CritPredecessorPath;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    unsigned int   Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
```

timecomp2.c

```
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinOut) != OUTPUT_E )
    return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);

MinSlack = GnlCritPathReqTime (CritSuccessorPath) -
            GnlCritPathArrTime (CritSuccessorPath);
SlackRise = GnlTimingInfoRequiredRise (TimingI) -
            GnlTimingInfoArrivalRise (TimingI);
SlackFall = GnlTimingInfoRequiredFall (TimingI) -
            GnlTimingInfoArrivalFall (TimingI);

if (!GnlFloatEqual (MinSlack, SlackRise, 4)
    && !GnlFloatEqual (MinSlack, SlackFall, 4) )
    return (GNL_OK);

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathArrTime (*CritPath,
                      GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath,
                      GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath,
                          GnlCritPathKindOfTime (CritSuccessorPath));
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                      GnlSequentialCompoInstName (SeqCompo));
SetGnlCritPathIncr (*CritPath, GnlCritPathArrTime (*CritPath));

return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFromInputOfSeqCell */
/*-----*/
GNL_STATUS GnlGetCritPathFromInputOfSeqCell (GNL_ASSOC          Assoc,
                                              LIBC_LIB      Lib,
                                              GNL_CRITICAL_PATH *CritPath,
                                              GNL_CRITICAL_PATH CritSuccessorPath,
                                              int             KindOfTime)
{
    int             i, Rank;
    char            *Formal;
    GNL_VAR         Var;
    GNL_TIMING_INFO TimingInfo;
    float           TimeRise, TimeFall, MinSlack;
    BLIST           Interface;
    GNL_STATUS      GnlStatus;
    GNL_CRITICAL_PATH CritPredecessorPath;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    int             IsCombPath, IsSeqPath;
```

timecomp2.c

```
unsigned int          Key;

Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);
SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathFanout (*CritPath, GnlTimingInfoFanout (TimingInfo));
SetGnlCritPathKindOfTime (*CritPath, KindOfTime);
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                      GnlSequentialCompoInstName (SeqCompo));

if (GnlStatus = GnlComputeSetupHoldTimeFromSeqCompo (SeqCompo, Formal,
                TimingInfo, &TimeRise, &TimeFall, Lib, 0))
    return (GnlStatus);
if (GnlGetCritPathFromVar (Var, Lib, &CritPredecessorPath,
                           *CritPath, KindOfTime, &IsCombPath, &IsSeqPath, 1, 0))
    return (GNL_MEMORY_FULL);

if (KindOfTime == RISE)
{
    SetGnlCritPathArrTime (*CritPath, GnlTimingInfoArrivalRise
(TimingInfo));
    SetGnlCritPathReqTime (*CritPath, GnlTimingInfoRequiredRise
(TimingInfo));
    SetGnlCritPathIncr (*CritPath, TimeRise);
}
else
{
    SetGnlCritPathArrTime (*CritPath, GnlTimingInfoArrivalFall
(TimingInfo));
    SetGnlCritPathReqTime (*CritPath, GnlTimingInfoRequiredFall
(TimingInfo));
    SetGnlCritPathIncr (*CritPath, TimeFall);
}

GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFrom3State                               */
/*-----*/
GNL_STATUS GnlGetCritPathFrom3State (GNL_ASSOC Assoc,
                                      LIBC_LIB Lib,
                                      GNL_CRITICAL_PATH *CritPath,
                                      GNL_CRITICAL_PATH CritSuccessorPath,
                                      int *IsCombPath, int *IsSeqPath)
```

timecomp2.c

```
{  
    LIBC_CELL          Cell;  
    LIBC_PIN           PinOut, PinIn;  
    int                i, Fanout, Rank, SensRise, SensFall;  
    GNL_ASSOC          AssocI;  
    char               *Formal;  
    GNL_VAR            Var, VarI;  
    GNL_TIMING_INFO   TimingI;  
    float              InTransRise, InTransFall, OutTransR, OutTransF,  
                      DelayR, DelayF, TimeRise, TimeFall;  
    float              Length, WireCapa, WireResistance, OutCapa;  
    float              SlackRise, SlackFall, MinSlack;  
    BLIST              Interface;  
    GNL_STATUS         GnlStatus;  
    int                KindOfTime;  
    GNL_CRITICAL_PATH CritPredecessorPath;  
    GNL_TRISTATE_COMPONENT TriStateCompo;  
    unsigned int        Key;  
  
    TriStateCompo = (GNL_TRISTATE_COMPONENT)GnlAssocTraversalInfoComponent  
(Assoc);  
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);  
    Var = GnlAssocActualPort (Assoc);  
    Formal = (char *) GnlAssocFormalPort (Assoc);  
  
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))  
        return (GNL_OK);  
    if (LibPinDirection (PinOut) == INPUT_E)  
        return (GNL_OK);  
  
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
                                                               &TimingI, &Key, &Rank)))  
        return (GnlStatus);  
  
    Fanout = GnlTimingInfoFanout (TimingI);  
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);  
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);  
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);  
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);  
  
    Interface = GnlTriStateInterface (TriStateCompo);  
    MinSlack = GnlCritPathReqTime (CritSuccessorPath) -  
               GnlCritPathArrTime (CritSuccessorPath);  
    for (i=0; i < BListSize (Interface); i++)  
    {  
        AssocI = (GNL_ASSOC)BListElt (Interface, i);  
        if (!AssocI)  
            continue;  
        VarI = GnlAssocActualPort (AssocI);  
        if (!VarI)  
            continue;  
        if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))  
            continue;  
        Formal = (char *) GnlAssocFormalPort (AssocI);  
        if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))  
            continue;
```

timecomp2.c

```
if (LibPinDirection(PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);
if (GnlVarDir (VarI) == GNL_VAR_INOUT)
    TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);

GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoArrivalRise (TimingI),
                                                GnlTimingInfoArrivalFall (TimingI),
                                                &TimeRise, &TimeFall);

if (!GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeRise,
4)
&& !GnlFloatEqual (GnlCritPathArrTime (CritSuccessorPath), TimeFall,
4))
    continue;

if ((GnlStatus = GnlCreateCritPath (CritPath)))
    return (GnlStatus);
CritPredecessorPath = NULL;
SetGnlCritPathCriticalAssoc (*CritPath, Assoc);
SetGnlCritPathFanout (*CritPath, Fanout);
SetGnlCritPathArrTime (*CritPath,
                      GnlCritPathArrTime (CritSuccessorPath));
SetGnlCritPathReqTime (*CritPath,
                      GnlCritPathReqTime (CritSuccessorPath));
SetGnlCritPathKindOfTime (*CritPath,
                          GnlCritPathKindOfTime (CritSuccessorPath));
GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
SetGnlCritPathInstNameFromUserCompo (*CritPath,
                                      GnlTriStateInstName (TriStateCompo));
GnlGetSensFromArcTimingAndInputArrivalTime (PinIn, PinOut,
                                             GnlTimingInfoArrivalRise (TimingI),
                                             GnlTimingInfoArrivalFall (TimingI),
                                             &SensRise,&SensFall);
if (GnlCritPathKindOfTime (CritSuccessorPath) == RISE)
{
    SetGnlCritPathIncr (*CritPath, DelayR);
    KindOfTime = SensRise;
}
else
{
    SetGnlCritPathIncr (*CritPath, DelayF);
    KindOfTime = SensFall;
}
```

```

        if (GnlStatus = GnlGetCritPathFromVar (VarI, Lib,
                                              &CritPredecessorPath,
                                              *CritPath, KindOfTime,
                                              IsCombPath, IsSeqPath, 1, 0))
            return (GnlStatus);
        GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
    }
    return (GNL_OK);
}
/*-----*/
/* GnlGetCritPathFromAssoc */
/*-----*/
GNL_STATUS GnlGetCritPathFromAssoc (GNL_ASSOC Assoc,
                                     LIBC_LIB Lib,
                                     GNL_CRITICAL_PATH *CritPath,
                                     GNL_CRITICAL_PATH CritSuccessorPath,
                                     int KindOfTime,
                                     int *IsCombPath, int *IsSeqPath)
{
    GNL_STATUS GnlStatus;
    int Rank;
    GNL_COMPONENT GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            if (GnlStatus = GnlGetCritPathFromSeqCell (Assoc, Lib,
                                                       CritPath, CritSuccessorPath))
                return (GnlStatus);
            *IsSeqPath = 1;
            break;

        case GNL_USER_COMPO:
            if (GnlStatus = GnlGetCritPathFromUserCompo (Assoc, Lib,
                                                       CritPath, CritSuccessorPath,
                                                       KindOfTime,
                                                       IsCombPath, IsSeqPath))
                return (GnlStatus);
            break;

        case GNL_TRISTATE_COMPO:
            if (GnlStatus = GnlGetCritPathFrom3State (Assoc, Lib,
                                                       CritPath, CritSuccessorPath,
                                                       IsCombPath, IsSeqPath))
                return (GnlStatus);
            break;

        case GNL_MACRO_COMPO:
            break;

        default:
            GnlError (12 /* Unknown component */);
            break;
    }
    return (GNL_OK);
}

```

timecomp2.c

```
}

/*-----*/
/* GnlGetCritPathFromVar */
/*-----*/
GNL_STATUS GnlGetCritPathFromVar (GNL_VAR Var,
                                  LIBC_LIB     Lib,
                                  GNL_CRITICAL_PATH *CritPath,
                                  GNL_CRITICAL_PATH CritSuccessorPath,
                                  int    KindOfTime,
                                  int *IsCombPath, int *IsSeqPath, int Start,
                                  int VarIsInout)
{
    GNL_VAR          VarI, Actual;
    GNL_ASSOC        AuxAssocVar;
    int             i, Rank;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO TimingInfo, TimingI;
    float           ArrivalTime, RequiredTime,
                    SlackRise, SlackFall, MinSlack;
    GNL_USER_COMPONENT InstOfGnl;
    GNL_CRITICAL_PATH CritPredecessorPath, CritPredecessorPath2;
    BLIST            AssocSources, RightVarAssigneds;
    unsigned int     Key;
    int              AuxVarIsInout;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    if (GnlVarDir (Var) == GNL_VAR_INOUT && Start)
        TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);

    if ((GnlStatus = GnlCreateCritPath (CritPath)))
        return (GnlStatus);
    CritPredecessorPath = NULL;
    SetGnlCritPathCriticalVar (*CritPath, Var);
    if (KindOfTime == RISE)
    {
        ArrivalTime = GnlTimingInfoArrivalRise (TimingInfo);
        RequiredTime = GnlTimingInfoRequiredRise (TimingInfo);
    }
    else
    {
        ArrivalTime = GnlTimingInfoArrivalFall (TimingInfo);
        RequiredTime = GnlTimingInfoRequiredFall (TimingInfo);
    }
    MinSlack = RequiredTime - ArrivalTime;
    SetGnlCritPathArrTime (*CritPath, ArrivalTime);
    SetGnlCritPathReqTime (*CritPath, RequiredTime);
    SetGnlCritPathKindOfTime (*CritPath, KindOfTime);
    GnlAddSuccessorToCritPath (*CritPath, CritSuccessorPath);
    SetGnlCritPathIncr (*CritPath, 0.0);

    if (GnlVarDir (Var) == GNL_VAR_INPUT ||
        (GnlVarDir (Var) == GNL_VAR_INOUT && Start))
    {
```

timecomp2.c

```
if (!BListSize (G_PileOfComponent))
{
    /*if (GnlVarDir (Var) == GNL_VAR_INPUT ||
        (GnlVarDir (Var) == GNL_VAR_INOUT && Start))*/
    {
        *IsCombPath = 1;
        return (GNL_OK);
    }
}
else
{
    InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock(Var,
InstOfGnl,&Actual);
    if (!AuxAssocVar)
    {
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }

    if (GnlTimingInfoCorrespToCurrentPathFromVar(Actual,
&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Actual) == GNL_VAR_INOUT && Start)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
    SlackRise = GnlTimingInfoRequiredRise (TimingI) -
                GnlTimingInfoArrivalRise (TimingI);
    SlackFall = GnlTimingInfoRequiredFall (TimingI) -
                GnlTimingInfoArrivalFall (TimingI);
    MinSlack = GnlCritPathReqTime (*CritPath) -
                GnlCritPathArrTime (*CritPath);
    if (!GnlFloatEqual (MinSlack, SlackRise, 4)
        && !GnlFloatEqual (MinSlack, SlackFall, 4) )
        return (GNL_OK);
    if ((GnlStatus = GnlCreateCritPath (&CritPredecessorPath)))
        return (GnlStatus);
    CritPredecessorPath2 = NULL;
    SetGnlCritPathCriticalAssoc (CritPredecessorPath, AuxAssocVar);
    SetGnlCritPathArrTime
(CritPredecessorPath,GnlCritPathArrTime(*CritPath));
    SetGnlCritPathReqTime
(CritPredecessorPath,GnlCritPathReqTime(*CritPath));
    SetGnlCritPathKindOfTime (CritPredecessorPath, KindOfTime);
    GnlAddSuccessorToCritPath (CritPredecessorPath, *CritPath);
    SetGnlCritPathIncr (CritPredecessorPath, 0.0);

    AuxVarIsInout = (GnlVarDir (Var) == GNL_VAR_INOUT );
    if (GnlGetCritPathFromVar (Actual, Lib, &CritPredecessorPath2,
                                CritPredecessorPath, KindOfTime,
                                IsCombPath, IsSeqPath, 1, AuxVarIsInout))
        return (GnlStatus);
    GnlAddPredecessorToCritPath (CritPredecessorPath,
CritPredecessorPath2);
```

timecomp2.c

```
GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
if (BListAddElt (G_PileOfComponent, InstOfGnl))
    return (GNL_MEMORY_FULL);
SetGnlCritPathIncr (CritPredecessorPath, 0.0);
return (GNL_OK);
}

}

/* in this section we compute the Successor critical Path from Var */

if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    VarI = (GNL_VAR) BListElt (RightVarAssigneds, i);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    SlackRise = GnlTimingInfoRequiredRise (TimingInfo) -
                GnlTimingInfoArrivalRise (TimingInfo);
    SlackFall = GnlTimingInfoRequiredFall (TimingInfo) -
                GnlTimingInfoArrivalFall (TimingInfo);
    if ( GnlFloatEqual (MinSlack, SlackRise, 4) ||
        GnlFloatEqual (MinSlack, SlackFall, 4) )
    {
        if (GnlGetCritPathFromVar (VarI, Lib, &CritPredecessorPath,
                                   *CritPath, KindOfTime,
                                   IsCombPath, IsSeqPath, 1, 0))
            return (GNL_MEMORY_FULL);
        GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
    }
}

BListQuickDelete (&RightVarAssigneds);

for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    if (GnlGetCritPathFromAssoc (AuxAssocVar, Lib, &CritPredecessorPath,
                                 *CritPath, KindOfTime,
                                 IsCombPath, IsSeqPath))
        return (GNL_MEMORY_FULL);
    GnlAddPredecessorToCritPath (*CritPath, CritPredecessorPath);
}

BListQuickDelete (&AssocSources);

return (GNL_OK);
}

/*-----*/
/* GnlGetMinSlackFromInputsOfSequentialInst */
/*-----*/
```

timecomp2.c

```

/* Doc procedure GnlGetMinSlackFromInputsOfSequentialInst :
   - Getting the slack minimal attached to the inputs of sequential instances
     of a given netlist (GNL) . */

/*-----*/
GNL_STATUS GnlGetMinSlackFromInputsOfSequentialInst (GNL Gnl, float
*MinSlack)
{
    GNL_VAR           Var;
    int              i, Rank;
    GNL_STATUS        GnlStatus;
    GNL_TIMING_INFO  TimingInfo;
    float             Slack;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT     ComponentI;
    GNL               GnlCompoI;
    unsigned int      Key;

for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
{
    ComponentI = (GNL_COMPONENT)BListElt (GnlComponents (Gnl), i);
    if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
    {
        GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
        if (GnlCompoI)
        {
            if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                return (GNL_MEMORY_FULL);
            if (GnlStatus = GnlGetMinSlackFromInputsOfSequentialInst (GnlCompoI,
                                                                      MinSlack))
                return (GnlStatus);
            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        }
    }
    if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
        continue;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
    Var = GnlSequentialCompoInput (SeqCompo);
    if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
        continue;
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

/*      fprintf (stderr,"%s\t%.2f\t%.2f\t%.2f\n", GnlSequentialCompoInstName
(SeqCompo), GnlTimingInfoArrivalRise (TimingInfo), GnlTimingInfoRequiredRise
(TimingInfo), GnlTimingInfoCapa (TimingInfo));
*/
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    if (*MinSlack > Slack)
        *MinSlack = Slack;

    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
}

```

timecomp2.c

```
if (*MinSlack > Slack)
    *MinSlack = Slack;

}

return (GNL_OK);
}

/*-----*/
/* GnlGetSequentialCritPathFromGnl */
/*-----*/
/* Doc procedure GnlGetSequentialCritPathFromGnl :
 - Getting the slack minimal attached to the inputs of sequential instances
   of a given netlist (GNL).

/*-----*/
GNL_STATUS GnlGetSequentialCritPathFromGnl (GNL Gnl, LIBC_LIB      Lib,
                                             BLIST ListOfCritPaths,
                                             float MinSlack)
{
    GNL_VAR           Var;
    int              i, Rank;
    GNL_STATUS        GnlStatus;
    GNL_TIMING_INFO  TimingInfo;
    float             Slack;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT     ComponentI;
    GNL               GnlCompoI;
    GNL_ASSOC         Assoc;
    GNL_CRITICAL_PATH CritPath;
    unsigned int      Key;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
            {
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                    return (GNL_MEMORY_FULL);
                if (GnlStatus = GnlGetSequentialCritPathFromGnl (GnlCompoI, Lib,
                                                               ListOfCritPaths, MinSlack))
                    return (GnlStatus);
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            }
        }
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
            continue;

        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
        Assoc = GnlSequentialCompoInputAssoc (SeqCompo);
        if (!Assoc)
            continue;
        Var = GnlAssocActualPort (Assoc);
        if (!Var)
```

timecomp2.c

```
    continue;
if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
    continue;
if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

Slack = GnlTimingInfoRequiredRise (TimingInfo) -
        GnlTimingInfoArrivalRise (TimingInfo);
if (MinSlack == Slack)
{
    if (GnlStatus = GnlGetCritPathFromInputOfSeqCell (Assoc, Lib, &CritPath,
                (GNL_CRITICAL_PATH)NULL, RISE))
        return (GnlStatus);
    if (BListAddElt (ListOfCritPaths, CritPath))
        return (GNL_MEMORY_FULL);
}
else
{
    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
        GnlTimingInfoArrivalFall (TimingInfo);
    if (MinSlack == Slack)
    {
        if (GnlStatus = GnlGetCritPathFromInputOfSeqCell (Assoc, Lib,
&CritPath,
                (GNL_CRITICAL_PATH)NULL, FALL))
            return (GnlStatus);
        if (BListAddElt (ListOfCritPaths, CritPath))
            return (GNL_MEMORY_FULL);
    }
}
return (GNL_OK);
}

/*-----
/* GnlGetMinSlackFromOutputs */
/*-----*/
/* Doc procedure GnlGetMinSlackFromOutputs :
 - Getting the slack minimal attached to the primary outputs of a given
 netlist (GNL).

/*-----*/
GNL_STATUS GnlGetMinSlackFromOutputs (GNL Gnl, float *MinSlack)
{
    GNL_VAR          Var;
    int              i, j, Rank;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float            Slack;
    BLIST            BucketI;
    unsigned int     Key;

*MinSlack = MAX_FLOAT;
for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
```

timecomp2.c

```
BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
for (j=0; j < BListSize (BucketI); j++)
{
    Var = (GNL_VAR) BListElt (BucketI, j);
    if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
        (GnlVarDir (Var) != GNL_VAR_INOUT)) )
        continue;
    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                    &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
                    GnlTimingInfoArrivalRise (TimingInfo);
    if (*MinSlack > Slack)
        *MinSlack = Slack;
    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                    GnlTimingInfoArrivalFall (TimingInfo);
    if (*MinSlack > Slack)
        *MinSlack = Slack;
}
return (GNL_OK);
}

/*-----*/
GNL_STATUS GnlGetCombinationalCritPathFromGnl (GNL      Gnl, LIBC_LIB      Lib,
                                              BLIST ListOfCritPaths,
                                              float   *MinSlackForCombPath,
                                              float   *MinSlackForSeqPath)
{
    GNL_VAR          Var;
    int              i, j, Rank, IsCombPath, IsSeqPath;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float            MinSlack, Slack;
    GNL_CRITICAL_PATH CritPath;
    BLIST            BucketI;
    unsigned int     Key;

/* GnlGetMinSlackFromOutputs (Gnl,&MinSlack); */
for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
            (GnlVarDir (Var) != GNL_VAR_INOUT)) )
            continue;

        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                        &TimingInfo, &Key, &Rank)))
            return (GnlStatus);

        IsCombPath = IsSeqPath = 0;
```

timecomp2.c

```
if (GnlTimingInfoArrivalRise (TimingInfo) > GnlTimingInfoArrivalFall (TimingInfo))
{
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    if (GnlStatus = GnlGetCritPathFromVar (Var, Lib, &CritPath,
                                           (GNL_CRITICAL_PATH) NULL, RISE,
                                           &IsCombPath, &IsSeqPath, 0, 0))
        return (GnlStatus);
}
else
{
    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo);
    if (GnlStatus = GnlGetCritPathFromVar (Var, Lib, &CritPath,
                                           (GNL_CRITICAL_PATH) NULL, FALL,
                                           &IsCombPath, &IsSeqPath, 0, 0))
        return (GnlStatus);
}

if (BListAddElt (ListOfCritPaths, CritPath))
    return (GNL_MEMORY_FULL);
if (IsCombPath)
    if (*MinSlackForCombPath > Slack)
        *MinSlackForCombPath = Slack;
if (IsSeqPath)
    if (*MinSlackForSeqPath > Slack)
        *MinSlackForSeqPath = Slack;
}

return (GNL_OK);
}

/*-----
/* GnlPrintHeadOfTimingReport
*/
/*-----*/
/* Doc procedure GnlPrintHeadOfTimingReport :
   - Printing global parameters used for timing analysis
*/
/*-----*/
GNL_STATUS GnlPrintHeadOfTimingReport (GNL          Gnl,
                                         LIBC_LIB     Lib)
{
    fprintf (stderr, " ----- \n");
    fprintf (stderr, " Report\t:t:\ttiming\n");
    fprintf (stderr, " max_number_paths\t:t:\t%d\n", GnlEnvPrintNbCritPath ());
    fprintf (stderr, " Design\t:t:\t%s\n", GnlName (Gnl));
    fprintf (stderr, " Date\t:t\t:t:\t");
    GnlPrintDate (stderr);
    fprintf (stderr, "\n");

    fprintf (stderr, " Operating Conditions\t:t:\t%s\n",
             LibOperCondOcName (G_OperCond));
    fprintf (stderr, " Library\t:t:\t%s\n", LibName (Lib));
}
```

timecomp2.c

```
fprintf (stderr, " Wire Loading Model\t:\t");
if (G_WireLoad)
    fprintf (stderr, "%s\n", LibWireLoadName (G_WireLoad));
else
    fprintf (stderr, "No Wire Load\n");
fprintf (stderr, " ----- \n");
fprintf (stderr, "\n");

}

/*-----
/* GnlGetCritPathFromNetwork
/*-----
/* Doc procedure GnlGetCritPathFromNetwork :
- Getting combinational critical path and Clock periode from a given
Network (GNL_NETWORK).
*/
/*-----
GNL_STATUS GnlGetCritPathFromNetwork (GNL_NETWORK      Nw,
                                         LIBC_LIB       Lib,
                                         int            MaxPaths,
                                         double         AreaWithoutNet,
                                         int            CellNumber,
                                         int            PrintClkDomain,
                                         int            PrintCritRegion)

{
    GNL           TopGnl;
    GNL_STATUS    GnlStatus;
    BLIST        ListOfCritPaths, ListOfSeqCritPaths;
    BLIST        AuxList, ListOfClocks;
    GNL_CRITICAL_PATH CritPath;
    int           NumberOfPaths, i, Size;
    float         MinSlack, ClockFreqValue, CombPathValue, MinSlackForSeqPath,
                  MinSlackForCombPath;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (CellNumber > MAX_CELL_NUMBER)
    {
        if (BListCreate (&G_PileOfComponent))
            return (GNL_MEMORY_FULL);

        if (BListCreate (&ListOfCritPaths))
            return (GNL_MEMORY_FULL);

        if (BListCreate (&ListOfSeqCritPaths))
            return (GNL_MEMORY_FULL);

        MinSlackForCombPath = MinSlackForSeqPath = MAX_FLOAT;
        if ((GnlStatus = GnlGetCombinationalCritPathFromGnl (TopGnl, Lib,
                                                               ListOfCritPaths,
                                                               &MinSlackForCombPath,
                                                               &MinSlackForSeqPath)))
            return (GnlStatus);
        NumberOfPaths = MaxPaths;
    }
}
```

timecomp2.c

```
GnlPrintHeadOfTimingReport (TopGnl, Lib);
ClockFreqValue = CombPathValue = 0.0;

for (i=0; i < BListSize (ListOfCritPaths); i++)
{
    CritPath = (GNL_CRITICAL_PATH) BListElt (ListOfCritPaths, i);
    if (BListCreate (&AuxList))
        return (GNL_MEMORY_FULL);
    Size = BListSize (ListOfSeqCritPaths);
    GnlPrintNEquivPathFromOneCritPath (CritPath, AuxList, &NumberOfPaths,
                                      0, &ClockFreqValue, &CombPathValue,
                                      MinSlackForSeqPath, MinSlackForCombPath,
                                      ListOfSeqCritPaths);

    if (BListSize (ListOfSeqCritPaths) != Size)
    {
        BListDelShift (ListOfCritPaths, i+1);
        i--;
    }

    BListQuickDelete (&AuxList);
    if (!NumberOfPaths)
        break;
}
BListDelete (&ListOfCritPaths, GnlFreeCritPath);

if (CombPathValue)
{
    fprintf (stderr, " -----\n");
    fprintf (stderr, " Combinational Path \t\t%.2f ns\n", CombPathValue);
    fprintf (stderr, " -----");
    fprintf (stderr, "\n\n");
}

MinSlack = MAX_FLOAT;
GnlGetMinSlackFromInputsOfSequentialInst (TopGnl, &MinSlack);
if (MinSlack < MinSlackForSeqPath)
{
    MinSlackForSeqPath = MinSlack;
    BListQuickDelete (&ListOfSeqCritPaths);
    if (BListCreate (&ListOfSeqCritPaths))
        return (GNL_MEMORY_FULL);
    if ((GnlStatus = GnlGetSequentialCritPathFromGnl (TopGnl, Lib,
                                                       ListOfSeqCritPaths, MinSlack)))
        return (GnlStatus);
}
else if (MinSlack == MinSlackForSeqPath)
{
    if ((GnlStatus = GnlGetSequentialCritPathFromGnl (TopGnl, Lib,
                                                       ListOfSeqCritPaths, MinSlack)))
        return (GnlStatus);
}

NumberOfPaths = MaxPaths;
ClockFreqValue = 0.0;
```

timecomp2.c

```

for (i=0; i < BListSize (ListOfSeqCritPaths); i++)
{
    CritPath = (GNL_CRITICAL_PATH) BListElt (ListOfSeqCritPaths, i);
    if (BListCreate (&AuxList))
        return (GNL_MEMORY_FULL);
    GnlPrintNEquivPathFromOneCritPath (CritPath, AuxList, &NumberOfPaths,
                                      1, &ClockFreqValue, &CombPathValue,
                                      MinSlackForSeqPath, MinSlackForCombPath,
                                      NULL);
    BListQuickDelete (&AuxList);
    if (!NumberOfPaths)
        break;
}
BListDelete (&ListOfSeqCritPaths, GnlFreeCritPath);
if (ClockFreqValue)
{
    fprintf (stderr, " -----
-----\n");
    fprintf (stderr, " Clock Period \t\t\t%.2f ns\n", ClockFreqValue);
    fprintf (stderr, " Clock Frequency \t\t\t%.2f Mhz\n",
            1000 * 1/ClockFreqValue);
    fprintf (stderr, " -----
-----\n");
}
BListQuickDelete (&ListOfCritPaths);
if (PrintCritRegion)
    if (GnlGetEpsiCriticInstancesFromGnl (TopGnl, Lib,
                                           MinSlackForSeqPath, MinSlackForCombPath, AreaWithoutNet))
        return (GNL_MEMORY_FULL);
BListQuickDelete (&G_PileOfComponent);
}
else
{
    if (BListCreate (&ListOfClocks))
        return (GNL_MEMORY_FULL);
    if (GnlStatus = GnlGetClocksFromNetwork (Nw, ListOfClocks,
PrintClkDomain))
        return (GnlStatus);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);
    GnlGetMinSlackFromOutputs (TopGnl,&MinSlackForCombPath);
    MinSlackForSeqPath = MAX_FLOAT;
    GnlGetMinSlackFromInputsOfSequentialInst (TopGnl, &MinSlackForSeqPath);
    BListQuickDelete (&G_PileOfComponent);
    if (GnlStatus = TimeExpandCritPathFromNetwork (Nw, Lib, MaxPaths,
                                                    ListOfClocks, AreaWithoutNet,
                                                    MinSlackForCombPath,
                                                    MinSlackForSeqPath,
                                                    PrintCritRegion))
        return (GnlStatus);
    BListDelete (&ListOfClocks, GnlFreeClock);
}
return (GNL_OK);
}

```

timecomp2.c

```
/*-----*
/* TimeOptByResizing
/*-----*/
GNL_STATUS TimeOptByResizing (GNL_NETWORK Nw,
                               LIBC_LIB      GnlLibc)
{
    double   AreaWithoutNet, AreaOfNets;
    int       CellNumber;
    GNL      TopGnl;

    TopGnl = GnlNetworkTopGnl(Nw);

    if (!GnlEnvRespectLibConstraints())
    {
        if (GnlLinkLib (Nw, TopGnl, GnlLibc))
            return (GNL_MEMORY_FULL);

        if (GnlSetListPathComponent (Nw))
            return (GNL_MEMORY_FULL);

        /* we resize the hash list of path component depending on the
        /* number of instances leading to a current Gnl. */
        if (GnlResizeHashListPathComponent (Nw))
            return (GNL_MEMORY_FULL);

        if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc))
            return (GNL_MEMORY_FULL);

        LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);

        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))
            return (GNL_MEMORY_FULL);
    }

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 0, &AreaWithoutNet, &AreaOfNets,
                               &CellNumber))
        return (GNL_MEMORY_FULL);

    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    fprintf (stderr, "\n");
    fprintf (stderr,
             " Optimization By Resizing Critical Instances...\n");
    fprintf (stderr, "\n");
    if (TimeOptByResizingFromGnl (TopGnl, GnlLibc))
        return (GNL_MEMORY_FULL);

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
                               &CellNumber))
        return (GNL_MEMORY_FULL);
```

timecomp2.c

```
fprintf (stderr, "\n");

    return (GNL_OK);
}

/*-----*/
/* GnlMeetLibConstraints */
/*-----*/
GNL_STATUS GnlMeetLibConstraints (GNL_NETWORK Nw,
                                  LIBC_LIB GnlLibc)
{
    double AreaWithoutNet, AreaOfNets;
    int CellNumber;
    GNL TopGnl;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (GnlLinkLib (Nw, TopGnl, GnlLibc))
        return (GNL_MEMORY_FULL);

    if (GnlSetListPathComponent (Nw))
        return (GNL_MEMORY_FULL);

    /* we resize the hash list of path component depending on the */
    /* number of instances leading to a current Gnl. */
    if (GnlResizeHashListPathComponent (Nw))
        return (GNL_MEMORY_FULL);

    if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc))
        return (GNL_MEMORY_FULL);

    LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);

    if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
                                &CellNumber))
        return (GNL_MEMORY_FULL);

    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    fprintf (stderr, "\n");
    fprintf (stderr,
             " Restructuring Netlist to meet Library Constraints...\n");
    fprintf (stderr, "\n");
    if (TimeLimitationFanoutFromNetwork (Nw, GnlLibc))
        return (GNL_MEMORY_FULL);

    CellNumber = 0;
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, 1, &AreaWithoutNet, &AreaOfNets,
```

timecomp2.c

```
        &CellNumber))  
    return (GNL_MEMORY_FULL);  
    fprintf (stderr, "\n");  
  
    return (GNL_OK);  
}  
  
/*-----*/  
/* TimeEstimate */  
/*-----*/  
GNL_STATUS TimeEstimate (GNL_NETWORK      Nw,  
                         LIBC_LIB   GnlLibc,  
                         int        PrintClkDomain,  
                         int        PrintCritRegion)  
{  
    double  AreaWithoutNet, AreaOfNets;  
    int     CellNumber;  
    GNL     TopGnl;  
    int     PrintData;  
  
    /* We call the construction of these data if:  
     * - we are in the ESTIMATION mode or  
     * - we are in another mode but we did not invoked the  
     *   control fanout. */  
    if ((GnlEnvMode () == GNL_MODE_ESTIMATION) ||  
        (!GnlEnvRespectLibConstraints() && !GnlEnvPostOpt () &&  
         (GnlEnvMode () != GNL_MODE_POST_OPTIMIZATION) ) )  
    {  
        TopGnl = GnlNetworkTopGnl (Nw);  
  
        if (GnlAddNetlistInfoForTraversalInComponent (Nw, TopGnl, GnlLibc))  
            return (GNL_MEMORY_FULL);  
  
        LibInitializeDeltaOperCondAndNomOperCond (GnlLibc, (char *)NULL);  
  
        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))  
            return (GNL_MEMORY_FULL);  
    }  
  
    if (GnlEnvPostOpt () || (GnlEnvMode () == GNL_MODE_POST_OPTIMIZATION) )  
        if (GnlComputeCapaAndFanoutFromNetwork (Nw, GnlLibc))  
            return (GNL_MEMORY_FULL);  
  
    CellNumber = 0;  
    PrintData = (!GnlEnvRespectLibConstraints() ||  
                (!GnlEnvPostOpt () &&  
                 (GnlEnvMode () != GNL_MODE_POST_OPTIMIZATION)) ||  
                (GnlEnvMode () == GNL_MODE_ESTIMATION) );  
    if (GnlGetAreaFromNetwork (Nw, GnlLibc, PrintData, &AreaWithoutNet,  
                               &AreaOfNets, &CellNumber))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlComputeArrivalTimeFromNetwork (Nw, GnlLibc))  
        return (GNL_MEMORY_FULL);
```

timecomp2.c

```
if (GnlComputeRequiredTimeFromNetwork (Nw, GnlLibc))
    return (GNL_MEMORY_FULL);

if (GnlGetCritPathFromNetwork (Nw, GnlLibc, GnlEnvPrintNbCritPath (),
                               AreaWithoutNet, CellNumber, PrintClkDomain,
                               PrintCritRegion))
    return (GNL_MEMORY_FULL);

if (GnlFreeTimingInfoFromNetwork (Nw))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/* ----- EOF ----- */
```

C:\Windows\Temp\GnlLibc\in\out\GnlEnvPrintNbCritPath.c

timefan.c

```
/*-----*/
/*
 *      File:          timefan.c
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description: *
 */
/*-----*/
/*-----*/
#include <stdio.h>

#include "blist.h"
#include "gnl.h"
#include "libc_mem.h"
#include "libc_api.h"
#include "gnlestim.h"
#include "time.h"
#include "bbdd.h"
#include "gnllibc.h"
#include "gnlmap.h"
#include "gnllibc.h"

#include "blist.e"
#include "libutil.e"
#include "timeutil.e"
#include "timecomp.e"
#include "timepath.e"
#include "timeutil.e"
#include "timefan.e"

extern GNL_STATUS GnlVarCreateAndAddInHashTableWithNoTest ();
extern GNL_STATUS GnlCreateUserComponent ();

/*-----*/
/* TimeGetListCapaPin */
/*-----*/
/* Doc procedure TimeGetListCapaPin :
   - Returns the List "ListCapaPin" of capacitance_pin from a List of Assoc.
   - Returns the List "ListFanout" of Fanout_pin from a List of Assoc.
*/
/*-----*/
GNL_STATUS TimeGetListCapaPin (BLIST AssocDest, GNL_VAR Var, BLIST
ListCapaPin,
                                BLIST ListFanout, BLIST ListRequiredTime,
                                LIBC_LIB Lib, int Tag)
{
    int             i, Rank;
    GNL_ASSOC      AssocI;
    GNL_COMPONENT  Compo;
    GNL_VAR        Vari, Formal;
    float          *Real, AuxCapa, Time, TimeRise, TimeFall;
    int             *Fanout;
```

timefan.c

```
LIBC_PIN Pin;
LIBC_KFATOR KF;
GNL_TIMING_INFO TimingI;
unsigned int Key;
GNL_VAR AuxFormal;
GNL_STATUS GnlStatus;
LIBC_CELL Cell;

for (i=0; i < BListSize (AssocDest); i++)
{
    AssocI = (GNL_ASSOC) BListElt (AssocDest, i);
    TimeRise = TimeFall = 0.0;
    if (GnlComputeRequiredTimeFromAssoc (AssocI, &TimeRise,
                                         &TimeFall, Lib, Var, Tag, 0))
        return (GNL_MEMORY_FULL);
    Time = TimeRise;
    if (Time > TimeFall)
        Time = TimeFall;

    Real = (float *) calloc (1, sizeof (float));
    (*Real) = Time;
    if (BListAddElt (ListRequiredTime, (int) Real))
        return (GNL_MEMORY_FULL);

    Compo = GnlAssocTraversalInfoComponent (AssocI);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlStatus = GnlGetCellFromGnlCompo (Compo, &Cell))
        return (GnlStatus);

    Formal = GnlAssocFormalPort (AssocI);
    if (Cell)
    {
        Pin = LibGetPinFromNameAndCell (Cell, (char*) Formal);
        Real = (float *) calloc (1, sizeof (float));
        Fanout = (int *) calloc (1, sizeof (int));
        AuxCapa = LibPinCapa (Pin);
        if (AuxCapa == 0.0)
            AuxCapa = LibDefaultInputPinCap (Lib);
        KF = LibKFactor (Lib);
        AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap (KF) [0],
                               LibKFactorPinCap (KF) [1],
                               LibKFactorPinCap (KF) [2]);

        *Real = AuxCapa;
        *Fanout = 1;
        if (BListAddElt (ListCapaPin, Real))
            return (GNL_MEMORY_FULL);
        if (BListAddElt (ListFanout, Fanout))
            return (GNL_MEMORY_FULL);
        continue;
    }

    if (GnlVarDir (Formal) == GNL_VAR_INPUT)
    {
        if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) Compo))
            return (GNL_MEMORY_FULL);
    }
}
```

timefan.c

```
if (GnlStatus = TimeGetFormalFromAssocAndActual
(AssocI, Var, &AuxFormal))
    return (GnlStatus);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
Real = (float *) calloc (1, sizeof (float));
Fanout = (int *) calloc (1, sizeof (int));
*Real = GnlTimingInfoCapa (TimingI);
*Fanout = GnlTimingInfoFanout (TimingI);
if (BListAddElt (ListCapaPin, Real))
    return (GNL_MEMORY_FULL);
if (BListAddElt (ListFanout, Fanout))
    return (GNL_MEMORY_FULL);
}

}

return (GNL_OK);
}

/*-----
/* TimeGetPinInAndPinOutOfBuffer
 */
void TimeGetPinInAndPinOutOfBuffer (LIBC_CELL Buffer, LIBC_PIN *PinIn,
                                    LIBC_PIN *PinOut)
{
    int      i;
    LIBC_PIN Pins;

    Pins = LibCellPins (Buffer);
    for (; Pins != NULL; Pins = LibPinNext (Pins))
    {
        if (LibPinDirection (Pins) == INPUT_E)
            *PinIn = Pins;
        if (LibPinDirection (Pins) == OUTPUT_E)
            *PinOut = Pins;
    }
}

/*-----
/* TimeGetInCapaAndLimitCapa
 */
void TimeGetInCapaAndLimitCapa (LIBC_CELL Buffer, float *InCapaBuffer,
                                 float *LimitCapaBuffer, LIBC_LIB Lib)
{
    int      i;
    LIBC_PIN PinIn, PinOut;
    float     AuxCapa;
    LIBC_KFATOR KF;

    *InCapaBuffer = *LimitCapaBuffer = 0.0;
    TimeGetPinInAndPinOutOfBuffer (Buffer, &PinIn, &PinOut);

    AuxCapa = LibPinCapa (PinIn);
```

timefan.c

```
if (AuxCapa == 0.0)
    AuxCapa = LibDefaultInputPinCap (Lib);
KF = LibKFactor (Lib);
AuxCapa = LibScalValue (AuxCapa, LibKFactorPinCap(KF) [0],
                        LibKFactorPinCap(KF) [1],
                        LibKFactorPinCap(KF) [2]);
*InCapaBuffer = AuxCapa;

AuxCapa = LibPinMaxCapa (PinOut);
if (AuxCapa == 0.0)
AuxCapa = LibDefaultMaxCapa (Lib);
*LimitCapaBuffer = AuxCapa;

}

/*-----
/* TimePrintPileOfComponent
/*-----*/
void TimePrintPileOfComponent (BLIST PileOfComponent)
{
    int      i;
    GNL_USER_COMPONENT    CompoI;

    for (i= 0; i < BListSize (PileOfComponent); i++)
    {
        CompoI = (GNL_USER_COMPONENT) BListElt (PileOfComponent, i);
        fprintf (stderr, "%s/", GnlUserComponentInstName (CompoI));
    }
    fprintf (stderr, "\n");
}
/*-----
/* TimeGetPileOfInstCompoFromPathCompo
/*-----*/
GNL_STATUS TimeGetPileOfInstCompoFromPathCompo (GNL_PATH_COMPONENT Path,
                                                BLIST PileOfInstCompo)
{
    GNL_PATH_COMPONENT    Previous;
    GNL_USER_COMPONENT    CompoI;

    if (!Path)
        return (GNL_OK);

    CompoI = GnlPathComponentComponent (Path);
    Previous = GnlPathComponentPrevious (Path);
    if (Previous)
        if (TimeGetPileOfInstCompoFromPathCompo (Previous, PileOfInstCompo))
            return (GNL_MEMORY_FULL);
    if (BListAddElt (PileOfInstCompo, CompoI))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
/*-----
/* TimeComputeCapaAndFanoutFromVar
/*-----*/
GNL_STATUS TimeUpdateCapaAndFanoutForNewVar (GNL_VAR Var, GNL_VAR NewVar,
                                              LIBC_LIB Lib, int *Tag,
                                              int InoutIsIn)
```

timefan.c

```
{  
    int i, j;  
    BLIST SubList, ListPaths;  
    GNL_PATH_COMPONENT Path;  
    BLIST SaveG_PileOfComponent;  
    BLIST PileOfInstCompo, AuxPileOfInstCompo;  
    GNL CurrentGnl;  
    GNL_USER_COMPONENT CurrentComponent;  
    char *HierNameOfGnlVar;  
    GNL_VAR SourceVar;  
  
/* unsigned int Key;  
   GNL_TIMING_INFO Timing;  
   int Rank;  
*/  
  
    SaveG_PileOfComponent = G_PileOfComponent;  
/* if (StringIdentical (GnlVarName (Var), "N576103"))  
{  
    fprintf (stderr, "G_PileOfComponent = ");  
    TimePrintPileOfComponent (G_PileOfComponent);  
    if (GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
        &Timing, &Key, &Rank))  
        return (GNL_MEMORY_FULL);  
    fprintf (stderr, "\t\tKey = %d\tRank = %d\tFanout = %d\tCapa = %.2f \n",  
        Key, Rank, GnlTimingInfoFanout (Timing), GnlTimingInfoCapa (Timing));  
}  
*/  
if (!BListSize (G_PileOfComponent))  
{  
    if (GnlComputeCapaAndFanoutFromVar (NewVar, Lib, *Tag, 1, 0))  
        return (GNL_MEMORY_FULL);  
    if (GnlGetSourceVarFromVar (Var, &SourceVar, &AuxPileOfInstCompo))  
        return (GNL_MEMORY_FULL);  
    G_PileOfComponent = AuxPileOfInstCompo;  
    if (!GnlVarIsVdd (SourceVar) && !GnlVarIsVss (SourceVar))  
        if (GnlComputeCapaAndFanoutFromVar (SourceVar, Lib, *Tag, InoutIsIn, 0))  
            return (GNL_MEMORY_FULL);  
    BListQuickDelete (&AuxPileOfInstCompo);  
    G_PileOfComponent = SaveG_PileOfComponent;  
    (*Tag)++;  
    return (GNL_OK);  
}  
  
    CurrentComponent = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,  
        BlistSize (G_PileOfComponent)-1);  
    CurrentGnl = GnlUserComponentGnlDef (CurrentComponent);  
    ListPaths = GnlListPathComponent (CurrentGnl);  
  
for (i=0; i < BlistSize (ListPaths); i++)  
{  
    SubList = (BLIST) BListElt (ListPaths, i);  
    if (!SubList)  
        continue;  
    for (j=0; j < BListSize (SubList); j++)  
    {  
        Path = (GNL_PATH_COMPONENT) BListElt (SubList, j);  
    }  
}
```

```

    if (BListCreate (&PileOfInstCompo))
        return (GNL_MEMORY_FULL);
    if (TimeGetPileOfInstCompoFromPathCompo (Path, PileOfInstCompo))
        return (GNL_MEMORY_FULL);
    G_PileOfComponent = PileOfInstCompo;
    if (GnlComputeCapaAndFanoutFromVar (NewVar, Lib, *Tag, 1, 0))
        return (GNL_MEMORY_FULL);
    if (GnlGetSourceVarFromVar (Var, &SourceVar, &AuxPileOfInstCompo))
        return (GNL_MEMORY_FULL);
    BListQuickDelete (&PileOfInstCompo);
    G_PileOfComponent = PileOfInstCompo = AuxPileOfInstCompo;
    if (!GnlVarIsVdd (SourceVar) && !GnlVarIsVss (SourceVar))
        if (GnlComputeCapaAndFanoutFromVar (SourceVar, Lib, *Tag,
InoutIsIn, 0))
            return (GNL_MEMORY_FULL);
    BListQuickDelete (&PileOfInstCompo);
}

}

(*Tag)++;
G_PileOfComponent = SaveG_PileOfComponent;
return (GNL_OK);
}

/*-----*/
/* TimeInsertBufferAtVar */
/*-----*/
GNL_STATUS TimeInsertBufferAtVar (GNL_VAR Var, LIBC_CELL Buffer, int K,
LIBC_LIB Lib, int *Id, GNL TopGnl,
int *Tag, int InoutIsIn)
{
    GNL_VAR_TRAVERSAL_INFO      TraversalInfo;
    GNL_ASSOC_TRAVERSAL_INFO    AssocTraversalInfo;
    BLIST                      AssocDests, LeftVarAssigneds, ListAssoc;
    GNL_ASSOC                  AssocIn, AssocOut, AssocI;
    GNL_VAR                     OutVar;
    char                        *InstanceName, *NameBuffer;
    BLIST                       Interface;
    LIBC_PIN                    PinIn, PinOut;
    int                          i;
    GNL_STATUS                  GnlStatus;
    GNL                         Gnl;
    GNL_USER_COMPONENT          InstOfGnl, UserComponent;
    char                        *Formal;

    Gnl = TopGnl;
    if (BListSize (G_PileOfComponent))
    {
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);
        Gnl = GnlUserComponentGnlDef (InstOfGnl);
    }
    ListAssoc = GnlVarTraversalInfoListAssoc (Var);
    if (GnlGetDestinationsOfVar (Var, &AssocDests, &LeftVarAssigneds))
        return (GNL_MEMORY_FULL);
}

```

timefan.c

```
/*
 if (GnlStrAppendIntCopy ("I_B_", *Id, &InstanceName))
     return (GNL_MEMORY_FULL);
*/
(*Id)++;

if (GnlStrCopy (LibCellName (Buffer), &NameBuffer))
    return (GNL_MEMORY_FULL);

if (BListCreateWithSize (2, &Interface))
    return (GNL_MEMORY_FULL);

TimeGetPinInAndPinOutOfBuffer (Buffer, &PinIn, &PinOut);

/* We create the Assoc attached to the input of the new buffer instance */
if (GnlCreateAssoc (&AssocIn))
    return (GNL_MEMORY_FULL);
if (GnlStrCopy (LibPinNameFirst (PinIn), &Formal))
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort (AssocIn, Formal);
SetGnlAssocActualPort (AssocIn, Var);
if (BListAddElt (Interface, (int) AssocIn))
    return (GNL_MEMORY_FULL);

/* We create the Assoc attached to the output of the new buffer instance */
if (GnlCreateAssoc (&AssocOut))
    return (GNL_MEMORY_FULL);
if (GnlStrCopy (LibPinNameFirst (PinOut), &Formal))
    return (GNL_MEMORY_FULL);
SetGnlAssocFormalPort (AssocOut, Formal);

if (GnlStatus = GnlCreateUniqueVar (Gnl, "\\$IB", &OutVar))
    return (GnlStatus);

if (GnlStrCopy (GnlVarName (OutVar), &InstanceName))
    return (GNL_MEMORY_FULL);

/*
 if (GnlStatus = GnlVarCreateAndAddInHashTableWithNoTest (Gnl, InstanceName,
                                                       &OutVar))
    return (GnlStatus);
*/
SetGnlAssocActualPort (AssocOut, OutVar);
if (BListAddElt (Interface, (int)AssocOut))
    return (GNL_MEMORY_FULL);

if (GnlCreateUserComponent (NameBuffer, InstanceName, (BLIST) NULL,
                           Interface, &UserComponent))
    return (GNL_MEMORY_FULL);
SetGnlUserComponentCellDef (UserComponent, Buffer);
if (GnlCreateAssocTraversalInfo (&AssocTraversalInfo))
    return (GNL_MEMORY_FULL);
SetGnlAssocHook (AssocIn, AssocTraversalInfo);
```

timefan.c

```
SetGnlAssocTraversalInfoComponent (AssocIn, (GNL_COMPONENT) UserComponent);
if (GnlCreateAssocTraversalInfo (&AssocTraversalInfo))
    return (GNL_MEMORY_FULL);
SetGnlAssocHook (AssocOut, AssocTraversalInfo);
SetGnlAssocTraversalInfoComponent (AssocOut, (GNL_COMPONENT)
UserComponent);

if (GnlCreateVarTraversalInfo (&TraversalInfo))
    return (GNL_MEMORY_FULL);
SetGnlVarHook (OutVar, TraversalInfo);

for (i= BListSize (AssocDests)-1; K >= 0; i--)
{
    AssocI = (GNL_ASSOC) BListElt (ListAssoc, i);
    BListDelShift (ListAssoc, i+1);
    SetGnlAssocActualPort (AssocI, OutVar);
    if (GnlUpdateVarAssocList (OutVar, AssocI))
        return (GNL_MEMORY_FULL);
    K--;
}
BListInsertInList (ListAssoc, AssocIn, 1);
BListQuickDelete (&LeftVarAssigneds);
BListQuickDelete (&AssocDests);
if (GnlUpdateVarAssocList (OutVar, AssocOut))
    return (GNL_MEMORY_FULL);

if (BListAddElt (GnlComponents (Gnl), (int) UserComponent))
    return (GNL_MEMORY_FULL);
if (TimeUpdateCapaAndFanoutForNewVar (Var, OutVar, Lib, Tag, InoutIsIn))
    return (GNL_MEMORY_FULL);
return (GNL_OK);
}

/*
 * TimeUpdateCapaAndFanoutOfVar
 */
void TimeUpdateCapaAndFanoutOfVar (GNL_VAR Var, int NewFanout,
                                    float NewCapa, int InoutIsIn)
{
    int             i, j;
    GNL_TIMING_INFO TimingI;
    BLIST           SubList;
    BLIST           ListTimingInfo;

    if (!BListSize (G_PileOfComponent))
    {
        TimingI = (GNL_TIMING_INFO) GnlVarTraversalInfoHook (Var);
        SetGnlTimingInfoFanout (TimingI, NewFanout);
        SetGnlTimingInfoCapa (TimingI, NewCapa);
        return;
    }

    ListTimingInfo = (BLIST) GnlVarTraversalInfoHook (Var);
    for (i=0; i < BListSize (ListTimingInfo); i++)
    {
        SubList = (BLIST) BListElt (ListTimingInfo, i);
```

```

if (!SubList)
    continue;

for (j=0; j < BListSize (SubList); j++)
{
    TimingI = (GNL_TIMING_INFO) BListElt (SubList, j);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook(TimingI);
    SetGnlTimingInfoFanout (TimingI, NewFanout);
    SetGnlTimingInfoCapa (TimingI, NewCapa);
}
}

/*-----*/
/* TimeLimitationFanoutOfAssoc */
/*-----*/
GNL_STATUS TimeLimitationFanoutOfAssoc (
    GNL_ASSOC          Assoc,
    GNL_VAR            Actual,
    float              LimitCapa,
    LIBC_CELL          Buffer,
    LIBC_LIB           Lib,
    int                *Id,
    GNL                 TopGnl,
    int                *Tag,
    BLIST              ListBuffers)
{
    GNL_COMPONENT      GnlCompo;
    GNL_VAR            Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO   TimingI;
    int                Rank;
    unsigned int        Key;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);
    if (GnlComponentType (GnlCompo) != GNL_USER_COMPO)
        return (GNL_OK);

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);
    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
        return (GNL_OK);

    if (BListAddElt (G_PileOfComponent, UserCompo))
        return (GNL_MEMORY_FULL);

    if (TimeGetFormalFromAssocAndActual (Assoc, Actual, &AuxFormal))
        return (GNL_MEMORY_FULL);

    if
(GnlTimingInfoCorrespToCurrentPathFromVar (AuxFormal, &TimingI, &Key, &Rank))
    return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT)
        TimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingI);
}

```

```

if (TimeLimitationFanoutOfVar (AuxFormal, LimitCapa, Buffer, Lib, Id,
                               TopGnl, Tag, 1, ListBuffers))
    return (GNL_MEMORY_FULL);

BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

return (GNL_OK);
}

/*-----
/* TimeLimitationFanoutOfVarDests
/*-----*/
/* Doc procedure TimeLimitationFanoutOfVarDests :
   - Controls the load capacitance of the Destinations of a net (GNL_VAR)
     taking into account the limit capacitance from the library.
*/
/*-----*/
GNL_STATUS TimeLimitationFanoutOfVarDests (
    GNL_VAR             Var,
    float               LimitCapa,
    LIBC_CELL           Buffer,
    LIBC_LIB            Lib,
    int                 *Id,
    int                 *ConstraintIsSatisfied,
    GNL                 TopGnl,
    int                 *Tag,
    int                 InoutIsIn,
    BLIST               ListBuffers)

{
    int                  i, Rank, N;
    GNL_VAR              AuxVar, Formal;
    float                AuxLimitCapa;
    float                Length, WireCapa, OutCapa;
    GNL_TIMING_INFO     AuxTimingI;
    unsigned int          Key;
    GNL_ASSOC             Assoc;
    GNL_COMPONENT         GnlCompo;
    GNL_USER_COMPONENT   UserCompo;
    BLIST                LeftVarAssigneds;
    BLIST                AssocDest;
    int                  Fanout, AuxId;

*ConstraintIsSatisfied = 0;
if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

if (!BListSize (AssocDest) && !BListSize (LeftVarAssigneds))
{
    BListQuickDelete (&AssocDest);
    BListQuickDelete (&LeftVarAssigneds);
    return (GNL_OK);
}
if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &AuxTimingI, &Key, &Rank))
    return (GNL_MEMORY_FULL);
if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
    AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);

```

```

Fanout = GnlTimingInfoFanout (AuxTimingI);
if (Fanout == 1)
{
    *ConstraintIsSatisfied = 1;
    return (GNL_OK);
}
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (AuxTimingI);

if (LimitCapa > OutCapa)
{
    *ConstraintIsSatisfied = 1;
    return (GNL_OK);
}
/* N = 0;
for (i=0; i < BListSize (AssocDest); i++)
{
    Assoc = (GNL_ASSOC) BListElt (AssocDest, i);
    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);
    if (GnlComponentType (GnlCompo) != GNL_USER_COMPO)
        continue;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    AuxVar = GnlAssocActualPort (Assoc);
    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
        continue;
    N++;
}
*/
AuxLimitCapa = LimitCapa;
/* if ((N != BListSize (AssocDest)) && BListSize (AssocDest))
   N++;
N += BListSize (LeftVarAssigneds);
if (N)
    AuxLimitCapa = LimitCapa / N;
*/
for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    AuxVar = (GNL_VAR) BListElt (LeftVarAssigneds, i);
    if (GnlTimingInfoCorrespToCurrentPathFromVar
(AuxVar, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    AuxId = *Id;
    if (TimeLimitationFanoutOfVar (AuxVar, AuxLimitCapa, Buffer, Lib, Id,
                                   TopGnl, Tag, 0, ListBuffers))
        return (GNL_MEMORY_FULL);

    if (AuxId == *Id)
        continue;

    if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad,

```

timefan.c

```
GnlTimingInfoFanout (AuxTimingI));
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (AuxTimingI);

if (LimitCapa > OutCapa)
{
    *ConstraintIsSatisfied = 1;
    return (GNL_OK);
}
}

for (i=0; i < BListSize (AssocDest); i++)
{
    Assoc = (GNL_ASSOC) BListElt (AssocDest, i);
    AuxId = *Id;
    if (TimeLimitationFanoutOfAssoc (Assoc, Var, AuxLimitCapa, Buffer, Lib,
Id,
                                         TopGnl, Tag, ListBuffers))
        return (GNL_MEMORY_FULL);
    if (AuxId == *Id)
        continue;

    if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad,
                                         GnlTimingInfoFanout (AuxTimingI));
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (AuxTimingI);

    if (LimitCapa > OutCapa)
    {
        *ConstraintIsSatisfied = 1;
        return (GNL_OK);
    }
}

return (GNL_OK);
}
/*-----*/
LIBC_CELL TimeSelectCorrectBufferDependingOfCapa (BLIST ListBuffers,
                                                float OutCapa, LIBC_LIB Lib)
{

    LIBC_CELL      Buffer;
    int           i, IndexCorrectBuffer;
    float          Diff;
    float          InCapaBuffer, LimitCapaBuffer;

    Diff = MAX_FLOAT;
    IndexCorrectBuffer = 0;
    for (i=0; i < BListSize (ListBuffers); i++)
    {
        Buffer = (LIBC_CELL) BListElt (ListBuffers, i);
        TimeGetInCapaAndLimitCapa (Buffer, &InCapaBuffer, &LimitCapaBuffer, Lib);
    }
}
```

timefan.c

```
if ((LimitCapaBuffer - OutCapa) < 0)
    continue;
if (Diff > (LimitCapaBuffer - OutCapa) )
{
    Diff = LimitCapaBuffer - OutCapa;
    IndexCorrectBuffer = i;
}
}
Buffer = (LIBC_CELL) BListElt (ListBuffers, IndexCorrectBuffer);
return (Buffer);
}

-----*/
void TimeComputeRequiredTimeAtBufferInput (LIBC_CELL Buffer,
                                            float OutCapa,
                                            int Fanout,
                                            float InTransRise,
                                            float InTransFall,
                                            LIBC_LIB Lib,
                                            float *RequiredTime)
{
    LIBC_PIN      PinIn, PinOut;
    float         WireResistance;
    float         DelayR, DelayF, OutTransR, OutTransF;
    float         Length;

    TimeGetPinInAndPinOutOfBuffer (Buffer, &PinIn, &PinOut);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Buffer, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);
    *RequiredTime = GnlMaxFloat (DelayR, DelayF);
}

-----*/
/*-----*/
GNL_STATUS TimeBalanceRequiredTime (GNL_VAR           Var,
                                    LIBC_CELL     Buffer,
                                    float        LimitCapa,
                                    LIBC_LIB     Lib,
                                    int          *Id,
                                    GNL          TopGnl,
                                    int          *Tag,
                                    int          InoutIsIn,
                                    BLIST        ListBuffers,
                                    int          *ConstraintIsSatisfied)
{
    BLIST          AssocDest;
    BLIST          LeftVarAssigneds, ListCapaPin, ListFanout;
    BLIST          ListRequiredTime;
    int            i, K, Stop, FanoutBuffer;
    int            AuxTag;
    float          LoadPinBuffer, LoadPinOfVar;
```

```

float          LoadWireBuffer, LoadWireVar;
float          Length, AuxFloat, WireCapa, OutCapa;
float          RequiredTimeI, RequiredTime;
GNL_TIMING_INFO AuxTimingI;
int            Rank, AuxInt, AuxId;
unsigned int   Key;
LIBC_CELL      CorrectBuffer;

*ConstraintIsSatisfied = 0;
if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar(Var, &AuxTimingI, &Key, &Rank))
    return (GNL_MEMORY_FULL);
if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
    AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
AuxTag = GnlTimingInfoTag (AuxTimingI);

if (BListCreateWithSize (BListSize (AssocDest), &ListCapaPin))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (BListSize (AssocDest), &ListRequiredTime))
    return (GNL_MEMORY_FULL);
if (BListCreateWithSize (BListSize (AssocDest), &ListFanout))
    return (GNL_MEMORY_FULL);
if (TimeGetListCapaPin (AssocDest, Var, ListCapaPin, ListFanout,
                        ListRequiredTime, Lib, AuxTag))
    return (GNL_MEMORY_FULL);

i = BListSize (ListCapaPin) - 1;
FanoutBuffer = 0;
LoadPinBuffer = 0.0;
K = -1;
Stop = 0;
while (!Stop)
{
    AuxFloat = *((float *) BListElt (ListCapaPin, i));
    AuxInt = *((int *) BListElt (ListFanout, i));
    RequiredTimeI = *((float *) BListElt (ListRequiredTime, i));
    FanoutBuffer = FanoutBuffer + AuxInt;
    LoadPinBuffer = LoadPinBuffer + AuxFloat;
    Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
    LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
                                                    Lib);
}

TimeComputeRequiredTimeAtBufferInput (Buffer,
                                      LoadPinBuffer+LoadWireBuffer,
                                      FanoutBuffer,
                                      GnlTimingInfoTransitionRise (AuxTimingI),
                                      GnlTimingInfoTransitionFall (AuxTimingI),
                                      Lib, &RequiredTime);
if ((RequiredTimeI - RequiredTime) >
    *((float*) BListElt (ListRequiredTime, 0)))
{
    K++;
    i--;
}
else

```

timefan.c

```
{  
    LoadPinBuffer = LoadPinBuffer - AuxFloat;  
    FanoutBuffer = FanoutBuffer - AuxInt;  
    Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);  
    LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,  
    Lib);  
    i = -1;  
}  
if (i == -1)  
    Stop = 1;  
}  
  
BListDelete (&ListCapaPin, BListElemFree);  
BListDelete (&ListRequiredTime, BListElemFree);  
BListDelete (&ListFanout, BListElemFree);  
BListQuickDelete (&AssocDest);  
BListQuickDelete (&LeftVarAssigneds);  
if (K >= 1)  
{  
    CorrectBuffer = TimeSelectCorrectBufferDependingOfCapa (ListBuffers,  
        LoadPinBuffer + LoadWireBuffer, Lib);  
    if (TimeInsertBufferAtVar (Var, CorrectBuffer, K, Lib, Id, TopGnl,  
        Tag, InoutIsIn))  
        return (GNL_MEMORY_FULL);  
    if (TimeBalanceRequiredTime (Var, Buffer, LimitCapa, Lib, Id,  
        TopGnl, Tag, InoutIsIn, ListBuffers,  
        ConstraintIsSatisfied))  
        return (GNL_MEMORY_FULL);  
    }  
  
if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &AuxTimingI, &Key, &Rank))  
    return (GNL_MEMORY_FULL);  
if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)  
    AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);  
LoadPinOfVar = GnlTimingInfoCapa (AuxTimingI);  
Length = LibGetWireLengthFromFanout (G_WireLoad,  
        GnlTimingInfoFanout (AuxTimingI));  
LoadWireVar = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);  
if ((LoadPinOfVar + LoadWireVar) <= LimitCapa)  
    *ConstraintIsSatisfied = 1;  
  
return (GNL_OK);  
}  
  
/*-----*/  
/* TimeLimitationFanoutOfVar */  
/*-----*/  
/* Doc procedure TimeLimitationFanoutOfVar :  
   - Controls the load capacitance of a net (GNL_VAR) taking into account  
     the limit capacitance from the library.  
*/  
/*-----*/  
  
extern GNL_STATUS TimeLimitationFanoutOfVar (GNL_VAR Var,  
    float LimitCapa,  
    LIBC_CELL Buffer,  
    LIBC_LIB Lib,
```

```

        int          *Id,
        GNL          TopGnl,
        int          *Tag,
        int          InoutIsIn,
        BLIST        ListBuffers)
{
    BLIST          AssocDest;
    BLIST          LeftVarAssigneds, ListCapaPin, ListFanout;
    BLIST          ListRequiredTime;
    int           i, K, Stop, FanoutBuffer, Fanout, FanoutVar;
    int           ConstraintIsSatisfied, AuxTag;
    GNL_ASSOC     Assoc;
    GNL_VAR       Actual;
    float         InCapaBuffer, LimitCapaBuffer;
    float         LoadPinBuffer, LoadPinOfVar;
    float         LoadWireBuffer, LoadWireVar;
    float         Length, AuxFloat, WireCapa, OutCapa;
    GNL_USER_COMPONENT InstOfGnl;
    char          *Str;
    GNL_TIMING_INFO AuxTimingI;
    int           Rank, AuxInt, AuxId;
    unsigned int   Key;
    LIBC_CELL     CorrectBuffer;

    if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar(Var, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
    AuxTag = GnlTimingInfoTag (AuxTimingI);
    Fanout = GnlTimingInfoFanout (AuxTimingI);
    if (Fanout == 1)
        return (GNL_OK);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (AuxTimingI);

    if (LimitCapa > OutCapa)
        return (GNL_OK);

    TimeGetInCapaAndLimitCapa (Buffer, &InCapaBuffer, &LimitCapaBuffer, Lib);
    if (InCapaBuffer >= LimitCapa)
        return (GNL_OK);

    if ( (GnlVarDir (Var) == GNL_VAR_OUTPUT) || (GnlVarDir (Var) ==
GNL_VAR_INOUT) )
    {
        if (BListSize (G_PileOfComponent))
        {
            InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                BListSize (G_PileOfComponent)-1);

            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            Assoc = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
&Actual);
        }
    }
}

```

```

if (!Assoc)
{
    if (BListAddElt (G_PileOfComponent, InstOfGnl))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
if (TimeLimitationFanoutOfVar (Actual, LimitCapa, Buffer, Lib,
                               Id, TopGnl, Tag, 0, ListBuffers))
    return (GNL_MEMORY_FULL);

if (BListAddElt (G_PileOfComponent, InstOfGnl))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar(Var, &AuxTimingI, &Key,
                                              &Rank))
    return (GNL_MEMORY_FULL);
if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
    AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
Fanout = GnlTimingInfoFanout (AuxTimingI);
if (Fanout == 1)
    return (GNL_OK);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (AuxTimingI);

if (LimitCapa > OutCapa)
    return (GNL_OK);
}

if (!BListSize (AssocDest))
{
    BListQuickDelete (&AssocDest);
    BListQuickDelete (&LeftVarAssigneds);
}
else
{
    ConstraintIsSatisfied = 0;

    if (TimeBalanceRequiredTime (Var, Buffer, LimitCapa, Lib, Id,
                                 TopGnl, Tag, InoutIsIn, ListBuffers,
                                 &ConstraintIsSatisfied))
        return (GNL_MEMORY_FULL);

    if (!ConstraintIsSatisfied)
    {
        BListQuickDelete (&AssocDest);
        BListQuickDelete (&LeftVarAssigneds);
        if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
            return (GNL_MEMORY_FULL);

        if (BListCreateWithSize (BListSize (AssocDest), &ListCapaPin))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (BListSize (AssocDest), &ListRequiredTime))
            return (GNL_MEMORY_FULL);
        if (BListCreateWithSize (BListSize (AssocDest), &ListFanout))

```

timefan.c

```
    return (GNL_MEMORY_FULL);
if (TimeGetListCapaPin (AssocDest, Var, ListCapaPin, ListFanout,
                        ListRequiredTime, Lib, AuxTag))
    return (GNL_MEMORY_FULL);

FanoutVar = GnlTimingInfoFanout (AuxTimingI);
i = BListSize (ListCapaPin) - 1;
FanoutBuffer = 0;
LoadPinBuffer = 0.0;
K = -1;
Stop = 0;
LoadPinOfVar = GnlTimingInfoCapa (AuxTimingI) + InCapaBuffer;
while (!Stop)
{
    AuxFloat = *((float *) BListElt (ListCapaPin, i));
    AuxInt = *((int *) BListElt (ListFanout, i));
    FanoutBuffer = FanoutBuffer + AuxInt;
    LoadPinBuffer = LoadPinBuffer + AuxFloat;
    Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
    LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad,
                                                    Length, Lib);

    LoadPinOfVar = LoadPinOfVar - AuxFloat;
    Length = LibGetWireLengthFromFanout (G_WireLoad,
                                         FanoutVar-FanoutBuffer+1);
    LoadWireVar = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
                                                Lib);

    if ((LoadPinBuffer + LoadWireBuffer) <= LimitCapaBuffer)
    {
        K++;
        i--;
    }
    else
    {
        LoadPinBuffer = LoadPinBuffer - AuxFloat;
        LoadPinOfVar = LoadPinOfVar + AuxFloat;
        FanoutBuffer = FanoutBuffer - AuxInt;
        Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
        LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad,
                                                        Length, Lib);
        Length = LibGetWireLengthFromFanout (G_WireLoad,
                                         FanoutVar-FanoutBuffer+1);
        LoadWireVar = LibGetWireScaledCapaFromLength (G_WireLoad,
                                                    Length, Lib);
        i = -1;
    }
    if (i == -1)
        Stop = 1;
    if ((LoadPinOfVar + LoadWireVar) <= LimitCapa)
        Stop = 1;
}

BListDelete (&ListCapaPin, BListElemFree);
BListDelete (&ListRequiredTime, BListElemFree);
BListDelete (&ListFanout, BListElemFree);
BListQuickDelete (&AssocDest);
```

```

BListQuickDelete (&LeftVarAssigneds);
if ( (K >= 0) && (InCapaBuffer < LoadPinBuffer))
{
    CorrectBuffer = TimeSelectCorrectBufferDependingOfCapa (ListBuffers,
        LoadPinBuffer + LoadWireBuffer, Lib);
    if (TimeInsertBufferAtVar (Var, CorrectBuffer, K, Lib, Id, TopGnl,
        Tag, InoutIsIn))
        return (GNL_MEMORY_FULL);
    if (TimeLimitationFanoutOfVar (Var, LimitCapa, Buffer, Lib, Id,
        TopGnl, Tag, InoutIsIn, ListBuffers))
        return (GNL_MEMORY_FULL);
    }
}
AuxId = *Id;
if (TimeLimitationFanoutOfVarDests (Var, LimitCapa, Buffer, Lib, Id,
    &ConstraintIsSatisfied, TopGnl, Tag,
    InoutIsIn, ListBuffers))
    return (GNL_MEMORY_FULL);

if ( (AuxId != *Id) && !ConstraintIsSatisfied)
    if (TimeLimitationFanoutOfVar (Var, LimitCapa, Buffer, Lib, Id,
        TopGnl, Tag, InoutIsIn, ListBuffers))
        return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----*/
/* TimeGetBestEquivCell */
/*-----*/
GNL_STATUS TimeGetBestEquivCell (GNL_USER_COMPONENT UserCompo,
    BLIST ListEquivCells,
    float OutCapa,
    LIB_DERIVE_CELL *BestCell,
    float *BestLimitCapa,
    LIBC_LIB Lib)
{
    int i;
    LIB_DERIVE_CELL DriveCell, CurrentDeriveCell, DeriveCellI;
    LIB_CELL MotherCell, CurrentMotherCell, MotherCellI;
    LIBC_CELL Cell;
    float LimitCapa;
    LIBC_PIN PinOut;
    int NegativeIndex, PositiveIndex;
    float NegativeDiff, PositiveDiff;
    float NegativeLimitCapa, PositiveLimitCapa;

    NegativeIndex = PositiveIndex = -1;
    NegativeDiff = -MAX_FLOAT;
    PositiveDiff = MAX_FLOAT;

    GnlFindCellsUserCompo (UserCompo, &CurrentMotherCell,
        &CurrentDeriveCell);
    for (i=0; i < BListSize (ListEquivCells); i++)
    {

```

timefan.c

```
DriveCell = (LIB_DERIVE_CELL) BListElt (ListEquivCells, i);
/* We dont accept the inverter cell */
if (LibDeriveCellBdd (CurrentDeriveCell) !=
    LibDeriveCellBdd (DriveCell))
    continue;

MotherCell = LibDeriveCellMotherCell (DriveCell);
Cell = LibHCellLibcCell (MotherCell);
PinOut = LibGetPinFromNameAndCell (Cell, LibHCellOutput (MotherCell));
LimitCapa = LibPinMaxCapa (PinOut);
if (LimitCapa == 0.0)
    LimitCapa = LibDefaultMaxCapa (Lib);

if ((LimitCapa - OutCapa) < 0)
{
    if (NegativeDiff < (LimitCapa - OutCapa))
    {
        NegativeDiff = (LimitCapa - OutCapa);
        NegativeIndex = i;
        NegativeLimitCapa = LimitCapa;
    }
}
else
if (PositiveDiff > (LimitCapa - OutCapa) )
{
    PositiveDiff = LimitCapa - OutCapa;
    PositiveIndex = i;
    PositiveLimitCapa = LimitCapa;
}
}

if (PositiveIndex >= 0)
{
    *BestCell = (LIB_DERIVE_CELL) BListElt (ListEquivCells, PositiveIndex);
    *BestLimitCapa = PositiveLimitCapa;
}
else
{
    *BestCell = (LIB_DERIVE_CELL) BListElt (ListEquivCells, NegativeIndex);
    *BestLimitCapa = NegativeLimitCapa;
}
return (GNL_OK);
}

/*-----
/* TimeReplaceByOtherEquivCell */
/*-----*/
GNL_STATUS TimeReplaceByOtherEquivCell (GNL_COMPONENT Compo,
                                         LIBC_LIB Lib,
                                         float          OutCapa,
                                         float          *BestLimitCapa,
                                         GNL           TopGnl)
{
    BLIST             ListEquivCells;
    LIB_DERIVE_CELL BestCell;
    GNL_USER_COMPONENT UserCompo;
```

```

switch (GnlComponentType (Compo)) {

    case GNL_USER_COMPO:
        UserCompo = (GNL_USER_COMPONENT) Compo;
        ListEquivCells = GnlUserComponentEquivCells (UserCompo);
        if (!ListEquivCells || !BListSize (ListEquivCells))
            return (GNL_OK);
        if (TimeGetBestEquivCell (UserCompo, ListEquivCells, OutCapa,
                                  &BestCell, BestLimitCapa, Lib))
            return (GNL_MEMORY_FULL);

        if (GnlReplaceUserCompoByDeriveCell (TopGnl, UserCompo,
                                              BestCell,
                                              (LIB_DERIVE_CELL) NULL,
                                              (BLIST) NULL))
            return (GNL_MEMORY_FULL);
        break;

    case GNL_SEQUENTIAL_COMPO:
        return (GNL_OK);

    case GNL_TRISTATE_COMPO:
        return (GNL_OK);

    case GNL_BUF_COMPO:
        return (GNL_OK);

    default:
        return (GNL_OK);
}

return (GNL_OK);
}

/*-----
/* TimeLimitationFanoutOfGenericCompo */
/*-----*/
GNL_STATUS TimeLimitationFanoutOfGenericCompo (GNL_COMPONENT Compo,
                                                LIBC_CELL Buffer,
                                                LIBC_LIB Lib,
                                                int *Id,
                                                GNL TopGnl,
                                                int *Tag,
                                                BLIST ListBuffers)
{
    LIBC_CELL Cell;
    LIBC_PIN PinOut;
    float LimitCapa, OutCapa, WireCapa, Length;
    BLIST Interface;
    GNL_ASSOC AssocI;
    int Rank, i, Fanout, J;
    char *Formal, *Str;
    GNL_VAR Vari, AuxVar;
    unsigned int Key;
    GNL_TIMING_INFO TimingI;
}

```

```

if (GnlGetCellFromGnlCompo (Compo, &Cell))
    return (GNL_MEMORY_FULL);

if (!Cell)
    return (GNL_OK);

if (GnlGetInterfaceFromGnlCompo (Compo, &Interface))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    if (!AssocI)
        continue;
    VarI = GnlAssocActualPort (AssocI);
    if (!VarI)
        continue;
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    if (GnlVarDir (VarI) == GNL_VAR_INOUT)
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinOut) != OUTPUT_E)
        continue;
    LimitCapa = LibPinMaxCapa (PinOut);

    if (GnlTimingInfoCorrespToCurrentPathFromVar
(Vari,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);

    if (StringIdentical (GnlVarName (VarI), "F376"))
        J=i;
    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    if (OutCapa > LimitCapa)
    {
        if (GnlGetHierarchycalInstName ((char *) NULL, &Str))
            return (GNL_MEMORY_FULL);
        /* fprintf (stderr,"-- Violation");
        if (Str)
            fprintf (stderr, "%s\t", Str);
        fprintf (stderr,"%s\t%.3f\t%.3f\t%d \n",
                GnlVarName (VarI), LimitCapa, OutCapa, Fanout);
        */
        /* if (TimeReplaceByOtherEquivCell (Compo, Lib, OutCapa,
        &LimitCapa,
                TopGnl))
            return (GNL_MEMORY_FULL);
        */
        if (TimeLimitationFanoutOfVar (Vari, LimitCapa, Buffer, Lib, Id,

```

```

        TopGnl, Tag, 0, ListBuffers))
    return (GNL_MEMORY_FULL);

}

}

return (GNL_OK);
}
/*-----
/*-----
/* TimeLimitationFanoutFromGnl
 */
/*-----*/
/* Doc procedure TimeLimitationFanoutFromGnl :
   - Satisfy or limits the fanout of each gate in a given
     Gnl (GNL).
   - It takes into account the values in the library
*/
/*-----*/
GNL_STATUS TimeLimitationFanoutFromGnl (GNL Gnl, LIBC_CELL Buffer, LIBC_LIB
Lib,
                                         int *Id, GNL TopGnl, int *Tag,
                                         BLIST ListBuffers)
{
    int
    GNL_STATUS
    GNL_COMPONENT
    GNL
    i;
    GnlStatus;
    ComponentI;
    GnlCompoI;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        if (!ComponentI)
            continue;
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
            {
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                    return (GNL_MEMORY_FULL);
                if (GnlStatus = TimeLimitationFanoutFromGnl (GnlCompoI, Buffer, Lib,
Id,
                                         TopGnl, Tag, ListBuffers))
                    return (GnlStatus);
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            }
        }
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
                continue;
        }
    }
}

```

timefan.c

```
if (TimeLimitationFanoutOfGenericCompo (ComponentI, Buffer, Lib,
                                         Id, TopGnl, Tag, ListBuffers))
    return (GNL_MEMORY_FULL);
}
return (GNL_OK);
}

/*-----*/
/* TimeLimitationFanoutFromNetwork . */
/*-----*/
/* Doc procedure TimeLimitationFanoutFromNetwork :
   - Satisfy or limits the fanout of each gate in a given
     Network (GNL_NETWORK).
   - It takes into account the values in the library
*/
/*-----*/
GNL_STATUS TimeLimitationFanoutFromNetwork (GNL_NETWORK      Nw,
                                              LIBC_LIB       Lib)
{
    GNL                  TopGnl;
    GNL_STATUS          GnlStatus;
    BLIST                ListBuffers;
    int                 i, Id, Tag;
    LIBC_CELL            CellI;
    LIBC_CELL            Buffer;
    float               MaxLimitCapaBuffer, InCapaBuffer, LimitCapaBuffer;
    GNL_LIB              HGnlLib;

    /* We pick up the internal lib representation. */
    HGnlLib = (GNL_LIB)LibHook (Lib);

    Id = 0;
    Tag = GnlNetworkTag (Nw) + 1;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    ListBuffers = GnlHLibCellsBuffers (HGnlLib);

    MaxLimitCapaBuffer = 0.0;

    Buffer = (LIBC_CELL) NULL;
    for (i=0; i < BListSize (ListBuffers); i++)
    {
        CellI = (LIBC_CELL) BListElt (ListBuffers, i);
        TimeGetInCapaAndLimitCapa (CellI, &InCapaBuffer, &LimitCapaBuffer, Lib);
        if (MaxLimitCapaBuffer < LimitCapaBuffer)
        {
            Buffer = CellI;
            MaxLimitCapaBuffer = LimitCapaBuffer;
        }
    }

    if (!BListSize (ListBuffers) || !Buffer)
```

timefan.c

```
{  
    return (GNL_OK);  
}  
  
if (TimeLimitationFanoutFromGnl (TopGnl, Buffer, Lib, &Id, TopGnl, &Tag,  
                                ListBuffers))  
    return (GNL_MEMORY_FULL);  
SetGnlNetworkTag (Nw, Tag);  
  
BListQuickDelete (&G_PileOfComponent);  
return (GNL_OK);  
}  
/*-----*/  
/*-----*/  
GNL_STATUS TimeOptBalanceRequiredTime (GNL_VAR Var,  
                                      LIBC_CELL Buffer,  
                                      LIBC_LIB Lib,  
                                      int *Id,  
                                      GNL TopGnl,  
                                      int *Tag,  
                                      int InoutIsIn,  
                                      BLIST ListBuffers)  
{  
    BLIST AssocDest;  
    BLIST LeftVarAssigneds, ListCapaPin, ListFanout;  
    BLIST ListRequiredTime;  
    int i, K, Stop, FanoutBuffer, Fanout;  
    int AuxTag;  
    float LoadPinBuffer, LoadPinOfVar;  
    float LoadWireBuffer, LoadWireVar;  
    float Length, AuxFloat, WireCapa, OutCapa;  
    float RequiredTimeI, RequiredTime;  
    GNL_TIMING_INFO AuxTimingI;  
    int Rank, AuxInt, AuxId;  
    unsigned int Key;  
    LIBC_CELL CorrectBuffer;  
  
    if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))  
        return (GNL_MEMORY_FULL);  
  
    if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &AuxTimingI, &Key, &Rank))  
        return (GNL_MEMORY_FULL);  
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)  
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);  
    Fanout = GnlTimingInfoFanout (AuxTimingI);  
    if (Fanout <= 3)  
        return (GNL_OK);  
    if (BListSize (AssocDest) <= 3 )  
        return (GNL_OK);  
  
    AuxTag = GnlTimingInfoTag (AuxTimingI);  
  
    if (BListCreateWithSize (BListSize (AssocDest), &ListCapaPin))  
        return (GNL_MEMORY_FULL);  
    if (BListCreateWithSize (BListSize (AssocDest), &ListRequiredTime))  
        return (GNL_MEMORY_FULL);  
    if (BListCreateWithSize (BListSize (AssocDest), &ListFanout))
```

timefan.c

```
    return (GNL_MEMORY_FULL);
if (TimeGetListCapaPin (AssocDest, Var, ListCapaPin, ListFanout,
                        ListRequiredTime, Lib, AuxTag))
    return (GNL_MEMORY_FULL);

i = BListSize (ListCapaPin) - 1;
FanoutBuffer = 0;
LoadPinBuffer = 0.0;
K = -1;
Stop = 0;
while (!Stop)
{
    AuxFloat = *((float *) BListElt (ListCapaPin, i));
    AuxInt = *((int *) BListElt (ListFanout, i));
    RequiredTimeI = *((float *) BListElt (ListRequiredTime, i));
    FanoutBuffer = FanoutBuffer + AuxInt;
    LoadPinBuffer = LoadPinBuffer + AuxFloat;
    Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
    LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
Lib);

    TimeComputeRequiredTimeAtBufferInput (Buffer,
                                         LoadPinBuffer+LoadWireBuffer,
                                         FanoutBuffer,
                                         GnlTimingInfoTransitionRise (AuxTimingI),
                                         GnlTimingInfoTransitionFall (AuxTimingI),
                                         Lib, &RequiredTime);
    if ((RequiredTimeI - RequiredTime) >
        *((float*) BListElt (ListRequiredTime, 0)))
    {
        K++;
        i--;
    }
    else
    {
        LoadPinBuffer = LoadPinBuffer - AuxFloat;
        FanoutBuffer = FanoutBuffer - AuxInt;
        Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
        LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
Lib);
        i = -1;
    }
    if (i == -1)
        Stop = 1;
}

BListDelete (&ListCapaPin, BListElemFree);
BListDelete (&ListRequiredTime, BListElemFree);
BListDelete (&ListFanout, BListElemFree);
BListQuickDelete (&AssocDest);
BListQuickDelete (&LeftVarAssigneds);
if (K >= 1)
{
/* CorrectBuffer = TimeSelectCorrectBufferDependingOfCapa (ListBuffers,
   LoadPinBuffer + LoadWireBuffer, Lib); */
if (TimeInsertBufferAtVar (Var, Buffer, K, Lib, Id, TopGnl,
                           Tag, InoutIsIn))
```

timefan.c

```

        return (GNL_MEMORY_FULL);
    if (TimeOptBalanceRequiredTime (Var, Buffer, Lib, Id,
                                   TopGnl, Tag, InoutIsIn, ListBuffers))
        return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}
/* ----- EOF ----- */

```

timefan.e

```
/*-----*/  
/*-----*/  
/*      File:          timefan.e           */  
/*      Version:        1.1                */  
/*      Modifications: -                 */  
/*      Documentation: -                */  
/*-----*/  
/*      Description:               */  
/*-----*/  
/*-----*/  
  
extern GNL_STATUS TimeLimitationFanoutOfVar (GNL_VAR           Var,  
                                              float            LimitCapa,  
                                              LIBC_CELL       Buffer,  
                                              LIBC_LIB        Lib,  
                                              int              *Id,  
                                              GNL             TopGnl,  
                                              int              *Tag,  
                                              int              InoutIsIn,  
                                              BLIST           ListBuffers);  
  
extern GNL_STATUS TimeLimitationFanoutFromNetwork (GNL_NETWORK      Nw,  
                                                 LIBC_LIB       Lib);  
  
extern GNL_STATUS TimeOptBalanceRequiredTime (GNL_VAR           Var,  
                                              LIBC_CELL       Buffer,  
                                              LIBC_LIB        Lib,  
                                              int              *Id,  
                                              GNL             TopGnl,  
                                              int              *Tag,  
                                              int              InoutIsIn,  
                                              BLIST           ListBuffers);  
  
/*-----*/  
/*-----*/  
/*-----*/  
/*-----*/
```

timeoptfan.c

```
/*-----*/
/*-----*/
GNL_STATUS TimeOptBalanceRequiredTime (GNL_VAR Var,
                                         LIBC_CELL Buffer,
                                         LIBC_LIB Lib,
                                         int *Id,
                                         GNL TopGnl,
                                         int *Tag,
                                         int InoutIsIn,
                                         BLIST ListBuffers)
{
    BLIST AssocDest;
    BLIST LeftVarAssigneds, ListCapaPin, ListFanout;
    BLIST ListRequiredTime;
    int i, K, Stop, FanoutBuffer, Fanout;
    int AuxTag;
    float LoadPinBuffer, LoadPinOfVar;
    float LoadWireBuffer, LoadWireVar;
    float Length, AuxFloat, WireCapa, OutCapa;
    float RequiredTimeI, RequiredTime;
    GNL_TIMING_INFO AuxTimingI;
    int Rank, AuxInt, AuxId;
    unsigned int Key;
    LIBC_CELL CorrectBuffer;

    if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar(Var, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
    Fanout = GnlTimingInfoFanout (AuxTimingI);
    if (Fanout <= 3)
        return (GNL_OK);
    if (BListSize (AssocDest) <= 3 )
        return (GNL_OK);

    AuxTag = GnlTimingInfoTag (AuxTimingI);

    if (BListCreateWithSize (BListSize (AssocDest), &ListCapaPin))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (BListSize (AssocDest), &ListRequiredTime))
        return (GNL_MEMORY_FULL);
    if (BListCreateWithSize (BListSize (AssocDest), &ListFanout))
        return (GNL_MEMORY_FULL);
    if (TimeGetListCapaPin (AssocDest, Var, ListCapaPin, ListFanout,
                           ListRequiredTime, Lib, AuxTag))
        return (GNL_MEMORY_FULL);

    i = BListSize (ListCapaPin) - 1;
    FanoutBuffer = 0;
    LoadPinBuffer = 0.0;
    K = -1;
    Stop = 0;
```

timeoptfan.c

```
while (!Stop)
{
    AuxFloat = *((float *) BListElt (ListCapaPin, i));
    AuxInt = *((int *) BListElt (ListFanout, i));
    RequiredTimeI = *((float *) BListElt (ListRequiredTime, i));
    FanoutBuffer = FanoutBuffer + AuxInt;
    LoadPinBuffer = LoadPinBuffer + AuxFloat;
    Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
    LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
    Lib);
    TimeComputeRequiredTimeAtBufferInput (Buffer,
                                         LoadPinBuffer+LoadWireBuffer,
                                         FanoutBuffer,
                                         GnlTimingInfoTransitionRise (AuxTimingI),
                                         GnlTimingInfoTransitionFall (AuxTimingI),
                                         Lib, &RequiredTime);
    if ((RequiredTimeI - RequiredTime) >
        *((float*) BListElt (ListRequiredTime, 0)))
    {
        K++;
        i--;
    }
    else
    {
        LoadPinBuffer = LoadPinBuffer - AuxFloat;
        FanoutBuffer = FanoutBuffer - AuxInt;
        Length = LibGetWireLengthFromFanout (G_WireLoad, FanoutBuffer);
        LoadWireBuffer = LibGetWireScaledCapaFromLength (G_WireLoad, Length,
    Lib);
        i = -1;
    }
    if (i == -1)
        Stop = 1;
}

BListDelete (&ListCapaPin, BListElemFree);
BListDelete (&ListRequiredTime, BListElemFree);
BListDelete (&ListFanout, BListElemFree);
BListQuickDelete (&AssocDest);
BListQuickDelete (&LeftVarAssigneds);
if (K >= 1)
{
    CorrectBuffer = TimeSelectCorrectBufferDependingOfCapa (ListBuffers,
                                                       LoadPinBuffer + LoadWireBuffer, Lib);
    if (TimeInsertBufferAtVar (Var, CorrectBuffer, K, Lib, Id, TopGnl,
                               Tag, InoutIsIn))
        return (GNL_MEMORY_FULL);
    if (TimeOptBalanceRequiredTime (Var, Buffer, Lib, Id,
                                   TopGnl, Tag, InoutIsIn, ListBuffers))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}
/*-----*/
/* TimeOptFanoutOfVar */
```

timeoptfan.c

```
/*-----*/
extern GNL_STATUS TimeOptFanoutOfVar (GNL_VAR Var,
                                       LIBC_CELL Buffer,
                                       LIBC_LIB Lib,
                                       int *Id,
                                       GNL TopGnl,
                                       int *Tag,
                                       int InoutIsIn,
                                       BLIST ListBuffers)
{
    BLIST AssocDests;
    BLIST LeftVarAssigneds, ListCapaPin, ListFanout;
    BLIST ListRequiredTime;
    int i, K, Stop, FanoutBuffer, Fanout, FanoutVar;
    int ConstraintIsSatisfied, AuxTag;
    GNL_ASSOC Assoc;
    GNL_VAR Actual;
    float InCapaBuffer, LimitCapaBuffer;
    float LoadPinBuffer, LoadPinOfVar;
    float LoadWireBuffer, LoadWireVar;
    float Length, AuxFloat, WireCapa, OutCapa;
    GNL_USER_COMPONENT InstOfGnl;
    char *Str;
    GNL_TIMING_INFO AuxTimingI;
    int Rank, AuxInt, AuxId;
    unsigned int Key;
    LIBC_CELL CorrectBuffer;

    if (GnlGetDestinationsOfVar (Var, &AssocDests, &LeftVarAssigneds))
        return (GNL_MEMORY_FULL);

    if (GnlTimingInfoCorrespToCurrentPathFromVar(Var, &AuxTimingI, &Key, &Rank))
        return (GNL_MEMORY_FULL);
    if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
        AuxTimingI = (GNL_TIMING_INFO) GnlTimingInfoHook (AuxTimingI);
    AuxTag = GnlTimingInfoTag (AuxTimingI);
    Fanout = GnlTimingInfoFanout (AuxTimingI);
    if (Fanout <= 3)
        return (GNL_OK);

    if (!BListSize (AssocDests))
    {
        BListQuickDelete (&AssocDests);
        BListQuickDelete (&LeftVarAssigneds);
    }
    else
    {
        if (TimeOptBalanceRequiredTime (Var, Buffer, Lib, Id,
                                       TopGnl, Tag, InoutIsIn, ListBuffers))
            return (GNL_MEMORY_FULL);
    }
    return (GNL_OK);
}
```

timepath.e

```
/*-----*/  
/*-----*/  
/*      File:          timepath.c  
/*      Version:        1.1  
/*      Modifications: -  
/*      Documentation: -  
/*  
/*      Description:           */  
/*-----*/  
  
#include <stdio.h>  
  
#include "blist.h"  
#include "gnl.h"  
#include "libc_mem.h"  
#include "libc_api.h"  
#include "gnlestim.h"  
#include "time.h"  
#include "bbdd.h"  
#include "gnllibc.h"  
#include "gnlmap.h"  
  
#include "blist.e"  
#include "libutil.e"  
#include "timeutil.e"  
#include "timecomp.e"  
#include "timepath.e"  
  
/* ----- */  
BLIST LocalListOfClocks;  
BLIST LocalListOfPathElem;  
BLIST MatrixCriticalPath;  
float LocalMinSlackForCombPath, LocalMinSlackForSeqPath;  
int G_Print = 0;  
int LocalIsAboutCombinational;  
  
#define HASH_LIST_PATH_ELEM    200  
/* ----- */  
/* TimeCreatePathElem           */  
/*-----*/  
GNL_STATUS TimeCreatePathElem (GNL_PATH_ELEM *PthElem)  
{  
  
    if ((*PthElem = (GNL_PATH_ELEM)  
          calloc (1, sizeof(GNL_PATH_ELEM_REC))) == NULL)  
        return (GNL_MEMORY_FULL);  
  
    SetGnlPathElemInPort (*PthElem, NULL);  
    SetGnlPathElemOutPort (*PthElem, NULL);;  
    SetGnlPathElemCompo (*PthElem, NULL);  
    SetGnlPathElemCriticalVar (*PthElem, NULL);  
    SetGnlPathElemFanout (*PthElem, 0);  
    SetGnlPathElemInstName (*PthElem, NULL);
```

timepath.e

```
SetGnlPathElemRiseArrTime (*PthElem, 0.0);
SetGnlPathElemFallArrTime (*PthElem, 0.0);
SetGnlPathElemRiseReqTime (*PthElem, 0.0);
SetGnlPathElemFallReqTime (*PthElem, 0.0);
SetGnlPathElemRiseIncr (*PthElem, 0.0);
SetGnlPathElemFallIncr (*PthElem, 0.0);

    return (GNL_OK);
}

#define OUT_END      199
#define FF_LATCH_END 200
#define ALL_END      201

/* -----
/* TimeFreePathElem
/*----- */
void TimeFreePathElem (GNL_PATH_ELEM *PthElem)
{

    if (!(*PthElem))
        return;

    if (GnlPathElemInPort (*PthElem))
        free (GnlPathElemInPort (*PthElem));
    if (GnlPathElemOutPort (*PthElem))
        free (GnlPathElemOutPort (*PthElem));
    if (GnlPathElemInstName (*PthElem))
        free (GnlPathElemInstName (*PthElem));

    *PthElem = NULL;
}

/* -----
/* -----
/* TimeCopyPathElem
/*----- */
int TimeConditionsAreSatisfied (int NbCp)
{
    BLIST          ListOfCritPaths, AuxList;
    float          MinSlack;
    GNL_PATH_ELEM  PthElem;
    int            Cond1, Cond2;

    if (BListSize (LocalListOfClocks) > 1 || NbCp > 1)
        return (0);
    Cond1 = Cond2 = 0;

    ListOfCritPaths = (BLIST) BListElt (
                                (BLIST) BListElt (MatrixCriticalPath, 0), 0);
    if (ListOfCritPaths)
    {
        AuxList = (BLIST) BListElt (ListOfCritPaths, 0);
        PthElem = (GNL_PATH_ELEM) BListElt (AuxList, BListSize (AuxList) - 1);
        MinSlack = GnlMinFloat (
                        GnlPathElemRiseReqTime (PthElem) - GnlPathElemRiseArrTime (PthElem),
                        GnlPathElemFallReqTime (PthElem) - GnlPathElemFallArrTime (PthElem));
    }
}
```

timepath.e

```
if (GnlFloatEqual (MinSlack, LocalMinSlackForCombPath, 4))
    Cond1 = 1;
}

if (BListSize (MatrixCriticalPath) >= 2)
{
    ListOfCritPaths = (BLIST) BListElt (
        (BLIST) BListElt (MatrixCriticalPath, 1), 1);
    if (ListOfCritPaths)
    {
        AuxList = (BLIST) BListElt (ListOfCritPaths, 0);
        PthElem = (GNL_PATH_ELEM) BListElt (AuxList, BListSize (AuxList) - 1);
        MinSlack = GnlMinFloat (
            GnlPathElemRiseReqTime (PthElem) - GnlPathElemRiseArrTime (PthElem),
            GnlPathElemFallReqTime (PthElem) - GnlPathElemFallArrTime (PthElem));
        if (GnlFloatEqual (MinSlack, LocalMinSlackForSeqPath, 4))
            Cond2 = 1;
    }
}
else
    Cond2 = 1;
if (LocalIsAboutCombinational)
    return (Cond1);
else
    return (Cond2);
/* return (Cond1 && Cond2); */
}
/*-----*/
/* TimeCopyPathElem */
/*-----*/
GNL_STATUS TimeCopyPathElem (GNL_PATH_ELEM PthElem1, GNL_PATH_ELEM *PthElem2)
{
    if (!PthElem1)
    {
        *PthElem2 = NULL;
        return (GNL_OK);
    }

    if (TimeCreatePathElem (PthElem2))
        return (GNL_MEMORY_FULL);

    if (GnlPathElemInPort (PthElem1))
        if (GnlStrCopy (GnlPathElemInPort
(PthElem1),&GnlPathElemInPort (*PthElem2)))
            return (GNL_MEMORY_FULL);
        if (GnlPathElemOutPort (PthElem1))
            if
(GnlStrCopy (GnlPathElemOutPort (PthElem1),&GnlPathElemOutPort (*PthElem2)))
                return (GNL_MEMORY_FULL);
        if (GnlPathElemInstName (PthElem1))

if (GnlStrCopy (GnlPathElemInstName (PthElem1),&GnlPathElemInstName (*PthElem2)))
    return (GNL_MEMORY_FULL);
    SetGnlPathElemRiseArrTime (*PthElem2, GnlPathElemRiseArrTime (PthElem1));
    SetGnlPathElemFallArrTime (*PthElem2, GnlPathElemFallArrTime (PthElem1));
    SetGnlPathElemRiseIncr (*PthElem2, GnlPathElemRiseIncr (PthElem1));
    SetGnlPathElemFallIncr (*PthElem2, GnlPathElemFallIncr (PthElem1));
```

timepath.e

```
SetGnlPathElemRiseReqTime (*PthElem2, GnlPathElemRiseReqTime (PthElem1));
SetGnlPathElemFallReqTime (*PthElem2, GnlPathElemFallReqTime (PthElem1));
SetGnlPathElemCompo (*PthElem2, GnlPathElemCompo (PthElem1));
SetGnlPathElemCriticalVar (*PthElem2, GnlPathElemCriticalVar (PthElem1));
SetGnlPathElemFanout (*PthElem2, GnlPathElemFanout (PthElem1));

    return (GNL_OK);
}
/* -----
/* TimeCopyPath
/* -----
/* Copy a Path (list of GNL_PATH_ELEM).
/* -----
GNL_STATUS TimeCopyPath (BLIST Path1, BLIST *Path2)
{
    int          i;
    GNL_PATH_ELEM    PthElem;

    if (!Path1)
    {
        *Path2 = NULL;
        return (GNL_OK);
    }

    if (BListCreate (Path2))
        return (GNL_MEMORY_FULL);

    for (i = 0; i < BListSize (Path1); i++)
    {
        if (TimeCopyPathElem ((GNL_PATH_ELEM) BListElt (Path1, i), &PthElem))
            return (GNL_MEMORY_FULL);

        if (BListAddElt (*Path2, PthElem))
            return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}
/* -----
/* TimeInsertPathInPathList
/* -----
GNL_STATUS TimeInsertPathInPathList (BLIST PathList, BLIST Path)
{
    int          i;
    GNL_PATH_ELEM    PthElem;
    float         MinFloat, AuxMinFloat;
    BLIST          AuxPath;

    PthElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path) - 1);
    MinFloat = GnlMinFloat (
        GnlPathElemRiseReqTime (PthElem) - GnlPathElemRiseArrTime (PthElem),
        GnlPathElemFallReqTime (PthElem) - GnlPathElemFallArrTime (PthElem));

    for (i=0; i < BListSize (PathList); i++)
    {
        AuxPath = (BLIST) BListElt (PathList, i);
        PthElem = (GNL_PATH_ELEM) BListElt (AuxPath, BListSize (AuxPath) - 1);
```

timepath.e

```
AuxMinFloat = GnlMinFloat (
    GnlPathElemRiseReqTime (PthElem) - GnlPathElemRiseArrTime (PthElem),
    GnlPathElemFallReqTime (PthElem) - GnlPathElemFallArrTime (PthElem));

    if (MinFloat < AuxMinFloat)
        break;
}
if (BListInsertInList (PathList, Path, i+1))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----*/
void TimeGetStartAndEndClockEndClockFromPath (BLIST Path, int *StartIndexClk,
                                              int *EndIndexClk)
{
    GNL_CLOCK Clock;
    int          i, Rank1, Rank2;
    BLIST         ListSeqHierInstNames;
    GNL_PATH_ELEM StartPthElem, EndPthElem;
    GNL_VAR       EndClockVar, StartClockVar;
    GNL_COMPONENT StartCompo, EndCompo;
    BLIST         SubList;
    unsigned int   Key;

*StartIndexClk = *EndIndexClk = 0;
StartPthElem = (GNL_PATH_ELEM) BListElt (Path, 0);
EndPthElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path) - 1);
Rank1 = Rank2 = 0;

StartCompo = GnlPathElemCompo (StartPthElem);
EndCompo = GnlPathElemCompo (EndPthElem);
if (!StartCompo && !EndCompo)
    return;

/* EndClockVar = NULL;
if (EndCompo)
    if (GnlComponentType (EndCompo) == GNL_SEQUENTIAL_COMPO)
        EndClockVar = (GNL_VAR) GnlSeqCompoInfoHook (
            (GNL_SEQUENTIAL_COMPONENT) EndCompo);
StartClockVar = NULL;
if (StartCompo)
    if (GnlComponentType (StartCompo) == GNL_SEQUENTIAL_COMPO)
        StartClockVar = (GNL_VAR) GnlSeqCompoInfoHook (
            (GNL_SEQUENTIAL_COMPONENT) StartCompo);
*/
for (i=0; i < BListSize (LocalListOfClocks); i++)
{
    Clock = (GNL_CLOCK) BListElt (LocalListOfClocks, i);
/*    if (StartClockVar == GnlClockSourceClock (Clock))
        *StartIndexClk = i+1;
    if (EndClockVar == GnlClockSourceClock (Clock))
        *EndIndexClk = i+1;
*/
    ListSeqHierInstNames = GnlClockSeqHierInstNames (Clock);
```

timepath.e

```
if (!Rank1 && GnlPathElemInstName (StartPthElem))
{
    Key = KeyOfName (GnlPathElemInstName (StartPthElem),
HASH_LIST_PATH_ELEM);
    SubList = (BLIST) BListElt (ListSeqHierInstNames, Key);
    Rank1 = BListMemberOfList (SubList,
        GnlPathElemInstName (StartPthElem), StringIdentical);
    if (Rank1)
    {
        *StartIndexClk = i+1;
    }
}

if (!Rank2 && GnlPathElemInstName (EndPthElem))
{
    Key = KeyOfName (GnlPathElemInstName (EndPthElem),
HASH_LIST_PATH_ELEM);
    SubList = (BLIST) BListElt (ListSeqHierInstNames, Key);
    Rank2 = BListMemberOfList (SubList,
        GnlPathElemInstName (EndPthElem), StringIdentical);
    if (Rank2)
    {
        *EndIndexClk = i+1;
    }
}

/*
-----*/
GNL_STATUS TimeUpdateListOfMostCritPath (int Clk1Index, int Clk2Index,
                                         BLIST Path, int NbCp)
{
    BLIST          AuxList, PathList;
    BLIST          AuxPath;

    AuxList = (BLIST) BListElt (MatrixCriticalPath, Clk1Index);
    PathList = (BLIST) BListElt (AuxList, Clk2Index);
    if (!PathList)
    {
        if (BListCreate (&PathList))
            return (GNL_MEMORY_FULL);
        BListElt (AuxList, Clk2Index) = (int) PathList;
    }
    if (TimeCopyPath (Path, &AuxPath))
        return (GNL_MEMORY_FULL);

    if (TimeInsertPathInPathList (PathList, AuxPath))
        return (GNL_MEMORY_FULL);

    if (BListSize (PathList) > NbCp)
    {
        AuxPath = (BLIST) BListElt (PathList, BListSize (PathList)-1);
        BListDelShift (PathList, BListSize (PathList));
        BListDelete (&AuxPath, TimeFreePathElem);
    }
    return (GNL_OK);
}
```

timepath.e

```
}

/*-----*/
GNL_STATUS TimeAddInListOfMostCritPath (BLIST Path, int NbCp)
{
    int          StartIndexClk, EndIndexClk;

    if (BListSize (Path) <= 1)
        return (GNL_OK);

    TimeGetStartAndEndClockEndClockFromPath (Path, &StartIndexClk,
&EndIndexClk);
    if (!StartIndexClk && !EndIndexClk)
    {
        /* This case is about combinational critical path */
        if (TimeUpdateListOfMostCritPath (0, 0, Path, NbCp))
            return (GNL_MEMORY_FULL);
    }
    else
    {
        if (StartIndexClk)
            if (TimeUpdateListOfMostCritPath(StartIndexClk, StartIndexClk, Path,
NbCp))
                return (GNL_MEMORY_FULL);

        if (EndIndexClk)
            if (StartIndexClk != EndIndexClk)
                if (TimeUpdateListOfMostCritPath(EndIndexClk, EndIndexClk, Path, NbCp))
                    return (GNL_MEMORY_FULL);

        if (EndIndexClk && StartIndexClk && (StartIndexClk != EndIndexClk) )
            if (TimeUpdateListOfMostCritPath(StartIndexClk, EndIndexClk, Path, 1))
                return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}

/*-----*/
/* TimePrintPath
 *-----*/
void TimePrintPath (BLIST Path)
{

    int          i, KindOfTime;
    GNL_PATH_ELEM          PthElem;
    float         MaxValue, MaxFloat, Setup;
    GNL_COMPONENT      Compo;
    LIBC_CELL       Cell;

    Setup = 0.0;
    fprintf (stderr, " Point\t\t\t\tFanout\tIncr\tPath\n");
    fprintf (stderr, " -----");
    --\n");
    for (i=0; i < BListSize (Path); i++)
    {
        PthElem = (GNL_PATH_ELEM) BListElt (Path, i);
```

timepath.e

```

MaxFloat = GnlMaxFloat (GnlPathElemRiseArrTime (PthElem),
                        GnlPathElemFallArrTime (PthElem));
if (MaxFloat == GnlPathElemRiseArrTime (PthElem))
    KindOfTime = RISE;
else
    KindOfTime = FALL;

if (i == BListSize (Path) - 1)
    MaxValue = MaxFloat;
if (!GnlPathElemCompo (PthElem))
{
    if (GnlPathElemInPort (PthElem))
        fprintf (stderr, " input external delay\t\t\t\tt0.0\tt0.0 \n");
    if (GnlPathElemInPort (PthElem))
        fprintf (stderr, " %s ", GnlPathElemInPort (PthElem));
    if (GnlPathElemOutPort (PthElem))
        fprintf (stderr, " %s ", GnlPathElemOutPort (PthElem));
    if (GnlPathElemInPort (PthElem))
        fprintf (stderr, "(in)\n\t\t\t\tt\t\t\tt");
    else
        fprintf (stderr, "(out)\n\t\t\t\tt\t\t\tt\t\tt");
    if (KindOfTime == RISE)
        fprintf (stderr, "% .3f\t% .3f Rising\n", GnlPathElemRiseIncr (PthElem),
                 GnlPathElemRiseArrTime (PthElem));
    else
        fprintf (stderr, "% .3f\t% .3f Falling\n", GnlPathElemFallIncr
(PthElem),
                 GnlPathElemFallArrTime (PthElem));
}
else
{
    Compo = GnlPathElemCompo (PthElem);
    GnlGetCellFromGnlCompo (Compo, &Cell);
    if (Cell)
    {
        fprintf (stderr, " %s", GnlPathElemInstName (PthElem));
        fprintf (stderr, " (%s)", LibCellName (Cell));
        if (GnlPathElemOutPort (PthElem))
        {
            if (GnlPathElemInPort (PthElem))
                fprintf (stderr, " (%s-->%s %s)\n\t\t\t\tt\t\t\tt",
                        GnlPathElemInPort (PthElem), GnlPathElemOutPort (PthElem),
                        GnlVarName (GnlPathElemCriticalVar (PthElem)));
            else
                fprintf (stderr, " (%s %s)\n\t\t\t\tt\t\t\tt\t\tt",
                        GnlPathElemOutPort (PthElem),
                        GnlVarName (GnlPathElemCriticalVar (PthElem)));
        }
        else
        {
            fprintf (stderr, " (%s %s)\n\t\t\t\tt\t\t\tt\t\tt",
                    GnlPathElemInPort (PthElem),
                    GnlVarName (GnlPathElemCriticalVar (PthElem)));
        }
    }
    if (KindOfTime == RISE)
        Setup = GnlPathElemRiseIncr (PthElem);
    else

```


timepath.e

```
    BListElt (LocalListOfPathElem, Key) = (int) SubList;
}

for (i=0; i < BListSize (SubList); i++)
{
    AuxPthElem = (GNL_PATH_ELEM) BListElt (SubList, i);
    if (StringIdentical (GnlPathElemInstName (AuxPthElem),
                         GnlPathElemInstName (*PathElem)))
    {
        AuxMax = GnlMaxFloat (GnlPathElemRiseArrTime (AuxPthElem),
                               GnlPathElemFallArrTime (AuxPthElem));
        if (Max <= AuxMax)
        {
            *Exist = 1;
            break;
        }
        else
        {
            TimeFreePathElem (&AuxPthElem);
            BListElt (SubList, i) = (int) *PathElem;
            break;
        }
    }
}

if (*Exist)
    TimeFreePathElem (PathElem);
else
if (i >= BListSize (SubList))
    if (BListAddElt (SubList, *PathElem))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}
/*-----*/
/*-----*/
void TimeFreeLocalListOfPathElem (TimeFreePathElem)
{
    int          i;
    BLIST        SubList;

    for (i=0; i < BListSize (LocalListOfPathElem); i++)
    {
        SubList = (BLIST) BListElt (LocalListOfPathElem, i);
        if (SubList)
            BListDelete (&SubList, TimeFreePathElem);
    }
    BListQuickDelete (&LocalListOfPathElem);
}

/*-----*/
GNL_STATUS TimeGetCritPathFromCombCell (BLIST Path, int NbCp,
                                         GNL_ASSOC Assoc, LIBC_LIB Lib,
                                         float RArrTime, float FArrTime,
                                         float InTransRise, float InTransFall)
{
    LIBC_CELL      Cell;
```

```

LIBC_PIN      PinOut, PinIn;
int           i, Fanout, Rank, Exist;
GNL_ASSOC     AssocI;
char          *Formal, *FormalI;
GNL_VAR       Var, VarI;
GNL_TIMING_INFO TimingI;
float         OutTransR, OutTransF,
              DelayR, DelayF, TimeRise, TimeFall;
float         Length, WireCapa, WireResistance, OutCapa;
BLIST         Interface;
GNL_STATUS    GnlStatus;
GNL_USER_COMPONENT UserCompo;
GNL_PATH_ELEM PathElem, AuxPathElem;
unsigned int   Key;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
Var = GnlAssocActualPort (Assoc);
FormalI = (char *) GnlAssocFormalPort (Assoc);

if (!(PinIn = LibGetPinFromNameAndCell (Cell, FormalI)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E)
    return (GNL_OK);

Interface = GnlUserComponentInterface (UserCompo);
for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinOut) != OUTPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                    Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                    Fanout, Cell, PinIn, PinOut,
                    &DelayR, &OutTransR,
                    &DelayF, &OutTransF);
}

```

timepath.e

```
GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                              DelayR, DelayF, RArrTime, FArrTime,
                                              &TimeRise, &TimeFall);

if (TimeCreatePathElem (&PathElem))
    return (GNL_MEMORY_FULL);
BListAddElt (Path, (int) PathElem);
if (GnlStrCopy (FormalI, &GnlPathElemInPort (PathElem)))
    return (GNL_MEMORY_FULL);
if (GnlStrCopy (Formal, &GnlPathElemOutPort (PathElem)))
    return (GNL_MEMORY_FULL);
if (GnlGetHierarchycalInstName (GnlUserComponentInstName (UserCompo),
                                 &GnlPathElemInstName (PathElem)))
    return (GNL_MEMORY_FULL);
SetGnlPathElemCompo (PathElem, GnlAssocTraversalInfoComponent
(Assoc));
SetGnlPathElemCriticalVar (PathElem, Vari);
SetGnlPathElemFanout (PathElem, Fanout);
SetGnlPathElemRiseArrTime (PathElem, TimeRise);
SetGnlPathElemFallArrTime (PathElem, TimeFall);
SetGnlPathElemRiseIncr (PathElem, DelayR);
SetGnlPathElemFallIncr (PathElem, DelayF);
SetGnlPathElemRiseReqTime (PathElem,
                           GnlTimingInfoRequiredRise(TimingI));
SetGnlPathElemFallReqTime (PathElem,
                           GnlTimingInfoRequiredFall(TimingI));
if (TimeExistPathElem (&PathElem, &Exist))
    return (GNL_MEMORY_FULL);
if (Exist)
{
    BListDelShift (Path, BListSize (Path));
    return (GNL_OK);
}
if (TimeGetCritPathFromVar (Path, NbCp, Vari, Lib, TimeRise,
                            TimeFall, OutTransR, OutTransF, 0))
    return (GNL_MEMORY_FULL);
BListDelShift (Path, BListSize (Path));
}
return (GNL_OK);
}

/*-----*/
GNL_STATUS TimeGetCritPathFrom3State (BLIST Path, int NbCp,
                                      GNL_ASSOC Assoc, LIBC_LIB Lib,
                                      float RArrTime, float FArrTime,
                                      float InTransRise, float InTransFall)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Fanout, Rank, Exist;
    GNL_ASSOC     AssocI;
    char          *Formal, *FormalI;
    GNL_VAR        Var, Vari;
    GNL_TIMING_INFO TimingI;
    float          OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall;
    float          Length, WireCapa, WireResistance, OutCapa;
```

timepath.e

```
BLIST           Interface;
GNL_STATUS      GnlStatus;
GNL_TRISTATE_COMPONENT TriState;
GNL_PATH_ELEM   PathElem, AuxPathElem;
unsigned int     Key;

TriState = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
Cell = GnlTriStateCompoInfoCell (TriState);
Var = GnlAssocActualPort (Assoc);
FormalI = (char *) GnlAssocFormalPort (Assoc);

if (!(PinIn = LibGetPinFromNameAndCell (Cell, FormalI)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E )
    return (GNL_OK);

Interface = GnlTriStateInterface (TriState);
for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    Vari = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (Vari) || GnlVarIsVdd (Vari))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinOut) != OUTPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Vari,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);

    Fanout = GnlTimingInfoFanout (TimingI);
    Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
    WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length,
                                                    Lib);
    WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
    OutCapa = WireCapa + GnlTimingInfoCapa (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                     Fanout, Cell, PinIn, PinOut,
                     &DelayR, &OutTransR,
                     &DelayF, &OutTransF);

    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                   DelayR, DelayF, RArrTime, FArrTime,
                                                   &TimeRise, &TimeFall);

    if (TimeCreatePathElem (&PathElem))
        return (GNL_MEMORY_FULL);
    BListAddElt (Path, (int) PathElem);
    if (GnlStrCopy (FormalI, &GnlPathElemInPort (PathElem)))
```

timepath.e

```
        return (GNL_MEMORY_FULL);
    if (GnlStrCopy (Formal, &GnlPathElemOutPort (PathElem)))
        return (GNL_MEMORY_FULL);
    if (GnlGetHierarchycalInstName (GnlTriStateInstName (TriState),
                                    &GnlPathElemInstName (PathElem)))
        return (GNL_MEMORY_FULL);
    SetGnlPathElemCompo (PathElem, GnlAssocTraversalInfoComponent
(Assoc));
    SetGnlPathElemCriticalVar (PathElem, VarI);
    SetGnlPathElemFanout (PathElem, Fanout);
    SetGnlPathElemRiseArrTime (PathElem, TimeRise);
    SetGnlPathElemFallArrTime (PathElem, TimeFall);
    SetGnlPathElemRiseIncr (PathElem, DelayR);
    SetGnlPathElemFallIncr (PathElem, DelayF);
    SetGnlPathElemRiseReqTime (PathElem,
                               GnlTimingInfoRequiredRise(TimingI));
    SetGnlPathElemFallReqTime (PathElem,
                               GnlTimingInfoRequiredFall(TimingI));

    if (TimeExistPathElem (&PathElem, &Exist))
        return (GNL_MEMORY_FULL);
    if (Exist)
    {
        BListDelShift (Path, BListSize (Path));
        return (GNL_OK);
    }
    if (TimeGetCritPathFromVar (Path, NbCp, VarI, Lib, TimeRise,
                                TimeFall, OutTransR, OutTransF, 0))
        return (GNL_MEMORY_FULL);
    BListDelShift (Path, BListSize (Path));
}
return (GNL_OK);
}

/*-----*/
GNL_STATUS TimeGetCritPathFromSeqCell (BLIST Path, int NbCp,
                                       GNL_ASSOC Assoc, LIBC_LIB Lib,
                                       float RArrTime, float FArrTime,
                                       float InTransRise, float InTransFall)
{
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    LIBC_CELL Cell;
    LIBC_PIN PinIn;
    int Rank;
    char *Formal;
    GNL_VAR Var;
    GNL_TIMING_INFO TimingI;
    GNL_STATUS GnlStatus;
    float TimeRise, TimeFall;
    GNL_PATH_ELEM PathElem, AuxPathElem;
    unsigned int Key;

    SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
    Cell = GnlSeqCompoInfoCell (SeqCompo);
    Var = GnlAssocActualPort (Assoc);
```

timepath.e

```
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinIn) == OUTPUT_E )
    return (GNL_OK);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);

if (GnlStatus = GnlComputeSetupHoldTimeFromSeqCompo (SeqCompo, Formal,
                TimingI, &TimeRise, &TimeFall, Lib, 0))
    return (GnlStatus);

if (TimeCreatePathElem (&PathElem))
    return (GNL_MEMORY_FULL);
BlistAddElt (Path, (int) PathElem);
if (GnlStrCopy (Formal, &GnlPathElemInPort (PathElem)))
    return (GNL_MEMORY_FULL);
if (GnlGetHierarchycalInstName (GnlSequentialCompoInstName (SeqCompo),
                &GnlPathElemInstName (PathElem)))
    return (GNL_MEMORY_FULL);
SetGnlPathElemCompo (PathElem, GnlAssocTraversalInfoComponent (Assoc));
SetGnlPathElemCriticalVar (PathElem, Var);
SetGnlPathElemFanout (PathElem, GnlTimingInfoFanout (TimingI));
SetGnlPathElemRiseArrTime (PathElem, RArrTime);
SetGnlPathElemFallArrTime (PathElem, FArrTime);
SetGnlPathElemRiseIncr (PathElem, TimeRise);
SetGnlPathElemFallIncr (PathElem, TimeFall);
SetGnlPathElemRiseReqTime (PathElem, GnlTimingInfoRequiredRise(TimingI));
SetGnlPathElemFallReqTime (PathElem, GnlTimingInfoRequiredFall(TimingI));
if (TimeAddInListOfMostCritPath (Path, NbCp))
    return (GNL_MEMORY_FULL);
AuxPathElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path)-1);
BlistDelShift (Path, BListSize (Path));
TimeFreePathElem (&AuxPathElem);

return (GNL_OK);
}
/*-----*/
GNL_STATUS TimeGetCritPathFromUserCompo (BLIST Path, int NbCp,
                GNL_ASSOC Assoc, LIBC_LIB Lib,
                float RArrTime, float FArrTime,
                float InTransRise, float InTransFall,
                GNL_VAR Actual)
{
    GNL_VAR          Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS        GnlStatus;
    int              Rank;

UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
Var = GnlAssocActualPort (Assoc);

Formal = GnlAssocFormalPort (Assoc);
```

timepath.e

```
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        if (GnlStatus = TimeGetCritPathFromCombCell (Path, NbCp, Assoc,
                                                    Lib, RArrTime, FArrTime,
                                                    InTransRise, InTransFall))
            return (GnlStatus);

        return (GNL_OK);
    }
    else
    /* It is a black box without any definition so we stop */
    /* at its boundary. */
    {
        fprintf (stderr,
                 " WARNING: black box <%s %s> may modify final estimation\n",
                 GnlUserComponentName (UserCompo),
                 GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
    return (GnlStatus);

if (GnlStatus = TimeGetCritPathFromVar (Path, NbCp, AuxFormal,
                                         Lib, RArrTime, FArrTime,
                                         InTransRise, InTransFall, 0))
    return (GnlStatus);

BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));

return (GNL_OK);
}
/*-----*/
GNL_STATUS TimeGetCritPathFromAssoc (BLIST Path, int NbCp,
                                      GNL_ASSOC Assoc, LIBC_LIB Lib,
                                      float RArrTime, float FArrTime,
                                      float InTransRise, float InTransFall,
                                      GNL_VAR Actual)
{
    GNL_STATUS      GnlStatus;
    int             Rank;
    GNL_COMPONENT   GnlCompo;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            if (GnlStatus = TimeGetCritPathFromSeqCell (Path, NbCp,
```

timepath.e

```
        Assoc, Lib, RArrTime, FArrTime,
        InTransRise, InTransFall))
    return (GnlStatus);
    break;

case GNL_USER_COMPO:
    if (GnlStatus = TimeGetCritPathFromUserCompo (Path, NbCp,
        Assoc, Lib, RArrTime, FArrTime,
        InTransRise, InTransFall, Actual))
    return (GnlStatus);
    break;

case GNL_TRISTATE_COMPO:
    if (GnlStatus = TimeGetCritPathFrom3State (Path, NbCp,
        Assoc, Lib, RArrTime, FArrTime,
        InTransRise, InTransFall))
    return (GnlStatus);
    break;

case GNL_MACRO_COMPO:
    break;

default:
    GnlError (12 /* Unknown component */);
    break;
}

return (GNL_OK);
}
/*-----*/
GNL_STATUS TimeGetCritPathFromVar (BLIST Path, int NbCp, GNL_VAR Var,
        LIBC_LIB Lib, float RArrTime, float FArrTime,
        float InTransRise, float InTransFall,
        int InoutIsIn)
{
    GNL_VAR          VarI, Actual;
    GNL_ASSOC        AuxAssocVar;
    char             *Name;
    int              i, Rank;
    GNL_STATUS       GnlStatus;
    GNL_USER_COMPONENT InstOfGnl;
    GNL_PATH_ELEM    PathElem, AuxPathElem;
    BLIST            AssocDest, LeftVarAssigneds;
    GNL_TIMING_INFO TimingI;
    unsigned int      Key;

    if (TimeConditionsAreSatisfied (NbCp))
        return (GNL_OK);

    if ((GnlVarDir (Var) == GNL_VAR_OUTPUT) || (GnlVarDir (Var) ==
GNL_VAR_INOUT) )
    {
        if (!BListSize (G_PileOfComponent))
        {
            if (GnlVarDir (Var) == GNL_VAR_OUTPUT ||
```

timepath.e

```
        (GnlVarDir (Var) == GNL_VAR_INOUT && !InoutIsIn))
    {
        if (TimeCreatePathElem (&PathElem))
            return (GNL_MEMORY_FULL);
        BListAddElt (Path, (int) PathElem);

        if (GnlStrCopy (GnlVarName (Var), &GnlPathElemOutPort (PathElem)))
            return (GNL_MEMORY_FULL);
        SetGnlPathElemCriticalVar (PathElem, Var);
        SetGnlPathElemRiseArrTime (PathElem, RArrTime);
        SetGnlPathElemFallArrTime (PathElem, FArrTime);
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                        &TimingI, &Key, &Rank)))
            return (GnlStatus);
        SetGnlPathElemFanout (PathElem, GnlTimingInfoFanout (TimingI));
        SetGnlPathElemRiseReqTime
(PathElem,GnlTimingInfoRequiredRise(TimingI));
        SetGnlPathElemFallReqTime
(PathElem,GnlTimingInfoRequiredFall(TimingI));
        if (TimeAddInListOfMostCritPath (Path, NbCp))
            return (GNL_MEMORY_FULL);
        AuxPathElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path)-1);
        BListDelShift (Path, BListSize (Path));
        TimeFreePathElem (&AuxPathElem);
        return (GNL_OK);
    }
}
else
{
    InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BListSize (G_PileOfComponent)-1);

    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
    AuxAssocVar
=GnlGetAssocFromFormalPortAndHierBlock(Var,InstOfGnl,&Actual);
    if (!AuxAssocVar)
    {
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
        return (GNL_OK);
    }
    if (TimeGetCritPathFromVar (Path, NbCp, Actual, Lib, RArrTime,
                                FArrTime, InTransRise, InTransFall, 0))
        return (GNL_MEMORY_FULL);
    if (BListAddElt (G_PileOfComponent, InstOfGnl))
        return (GNL_MEMORY_FULL);
    }
}

if (GnlGetDestinationsOfVar (Var, &AssocDest, &LeftVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (LeftVarAssigneds); i++)
{
    VarI = (GNL_VAR) BListElt (LeftVarAssigneds, i);

    if (TimeGetCritPathFromVar (Path, NbCp, VarI, Lib, RArrTime, FArrTime,
```

timepath.e

```
        InTransRise, InTransFall, 0))
    return (GNL_MEMORY_FULL);
}
BListQuickDelete (&LeftVarAssigneds);

for (i=0; i < BListSize (AssocDest); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocDest, i);
    if (TimeGetCritPathFromAssoc (Path, NbCp, AuxAssocVar, Lib, RArrTime,
                                  FArrTime, InTransRise, InTransFall, Var))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&AssocDest);

return (GNL_OK);
}

/*-----*/
GNL_STATUS TimeExpandCombinationalCritPathFromGnl (GNL Gnl, LIBC_LIB Lib,
                                                    int NbCp)
{
    GNL_VAR          Var;
    int              i, j, Rank;
    GNL_STATUS       GnlStatus;
    BLIST            Path;
    BLIST            BucketI;
    GNL_TIMING_INFO TimingInfo;
    GNL_PATH_ELEM    PathElem, AuxPathElem;
    float            ArrTimeR, ArrTimeF, InTransR, InTransF;
    unsigned int     Key;

    if (BListCreate (&Path))
        return (GNL_MEMORY_FULL);

    for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
    {
        BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
        for (j=0; j < BListSize (BucketI); j++)
        {
            Var = (GNL_VAR) BListElt (BucketI, j);
            if (!Var || (GnlVarDir (Var) != GNL_VAR_INPUT && GnlVarDir (Var) != GNL_VAR_INOUT))
                continue;
            if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                                      &TimingInfo, &Key, &Rank)))
                return (GnlStatus);
            if (GnlVarDir (Var) == GNL_VAR_INOUT)
                TimingInfo = (GNL_TIMING_INFO) GnlTimingInfoHook (TimingInfo);
            ArrTimeR = GnlTimingInfoArrivalRise (TimingInfo);
            ArrTimeF = GnlTimingInfoArrivalFall (TimingInfo);
            InTransR = GnlTimingInfoTransitionRise (TimingInfo);
            InTransF = GnlTimingInfoTransitionFall (TimingInfo);
            if (GnlVarDir (Var) == GNL_VAR_INOUT)
            {
```

timepath.e

```
    ArrTimeR = ArrTimeF = InTransR = InTransF = 0.0;
}
if (TimeCreatePathElem (&PathElem))
    return (GNL_MEMORY_FULL);
BListAddElt (Path, (int) PathElem);
if (GnlStrCopy (GnlVarName (Var), &GnlPathElemInPort (PathElem)))
    return (GNL_MEMORY_FULL);
SetGnlPathElemCriticalVar (PathElem, Var);
SetGnlPathElemFanout (PathElem, GnlTimingInfoFanout (TimingInfo));
SetGnlPathElemRiseArrTime (PathElem, ArrTimeR);
SetGnlPathElemFallArrTime (PathElem, ArrTimeF);
SetGnlPathElemRiseReqTime
(PathElem, GnlTimingInfoRequiredRise (TimingInfo));
    SetGnlPathElemFallReqTime
(PathElem, GnlTimingInfoRequiredFall (TimingInfo));
    if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &LocalListOfPathElem))
        return (GNL_MEMORY_FULL);
    BSize (LocalListOfPathElem) = HASH_LIST_PATH_ELEM;
    if (TimeGetCritPathFromVar (Path, NbCp, Var, Lib, ArrTimeR,
                                ArrTimeF, InTransR, InTransF, 1))
        return (GNL_MEMORY_FULL);
    TimeFreeLocalListOfPathElem (TimeFreePathElem);
    AuxPathElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path)-1);
    BListDelShift (Path, BListSize (Path));
    TimeFreePathElem (&AuxPathElem);
    if (TimeConditionsAreSatisfied (NbCp))
        break;
}
if (TimeConditionsAreSatisfied (NbCp))
    break;
}
BListQuickDelete (&Path);
return (GNL_OK);

}

/*-----
/* TimeExpandCritPathFromOutputOfSeqCell
*/
-----*/
GNL_STATUS TimeExpandCritPathFromOutputOfSeqCell (BLIST Path, int MaxPaths,
                                                GNL_ASSOC Assoc, LIBC_LIB Lib)
{
    int             Rank;
    char            *Formal;
    GNL_VAR         Var, AuxVar;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS      GnlStatus;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_PATH_ELEM   PathElem, AuxPathElem;
    GNL_ASSOC       AuxAssoc;
    unsigned int     Key;

    if (!Assoc)
        return (GNL_OK);
    Var = GnlAssocActualPort (Assoc);
    if (!Var)
        return (GNL_OK);
```

timepath.e

```
Formal = (char *) GnlAssocFormalPort (Assoc);
SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);

if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                &TimingInfo, &Key, &Rank)))
    return (GnlStatus);

if (TimeCreatePathElem (&PathElem))
    return (GNL_MEMORY_FULL);
BListAddElt (Path, (int) PathElem);
if (GnlStrCopy (Formal, &GnlPathElemOutPort (PathElem)))
    return (GNL_MEMORY_FULL);
/* AuxAssoc = (GNL_ASSOC) BListElt (GnlSequentialCompoInterface (SeqCompo),
2);
if (AuxAssoc && (AuxVar = GnlAssocActualPort (AuxAssoc)) &&
    !GnlVarIsVss (AuxVar) && !GnlVarIsVdd (AuxVar))
{
    Formal = (char *) GnlAssocFormalPort (AuxAssoc);
    if (Formal)
    {
        if (GnlStrCopy (Formal, &GnlPathElemInPort (PathElem)))
            return (GNL_MEMORY_FULL);
        SetGnlPathElemRiseIncr (PathElem,
GnlTimingInfoArrivalRise(TimingInfo));
        SetGnlPathElemFallIncr (PathElem,
GnlTimingInfoArrivalFall(TimingInfo));
    }
}
if (GnlGetHierarchicalInstName (GnlSequentialCompoInstName (SeqCompo),
&GnlPathElemInstName (PathElem)))
    return (GNL_MEMORY_FULL);
SetGnlPathElemCompo (PathElem, GnlAssocTraversalInfoComponent (Assoc));
SetGnlPathElemCriticalVar (PathElem, Var);
SetGnlPathElemFanout (PathElem, GnlTimingInfoFanout (TimingInfo));
SetGnlPathElemRiseArrTime (PathElem, GnlTimingInfoArrivalRise(TimingInfo));
SetGnlPathElemFallArrTime (PathElem, GnlTimingInfoArrivalFall(TimingInfo));
SetGnlPathElemRiseReqTime (PathElem,
GnlTimingInfoRequiredRise(TimingInfo));
SetGnlPathElemFallReqTime (PathElem,
GnlTimingInfoRequiredFall(TimingInfo));
if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &LocalListOfPathElem))
    return (GNL_MEMORY_FULL);
BSize (LocalListOfPathElem) = HASH_LIST_PATH_ELEM;
if (TimeGetCritPathFromVar (Path, MaxPaths, Var, Lib,
                GnlTimingInfoArrivalRise (TimingInfo),
                GnlTimingInfoArrivalFall (TimingInfo),
                GnlTimingInfoTransitionRise (TimingInfo),
                GnlTimingInfoTransitionFall (TimingInfo), 0))
    return (GNL_MEMORY_FULL);
TimeFreeLocalListOfPathElem (TimeFreePathElem);
AuxPathElem = (GNL_PATH_ELEM) BListElt (Path, BListSize (Path)-1);
BListDelShift (Path, BListSize (Path));
TimeFreePathElem (&AuxPathElem);

return (GNL_OK);
```

timepath.e

```
}

/*-----*/
/* TimeInsertSeqElemInSortedListOfSeqElem */
/*-----*/
GNL_STATUS TimeInsertSeqElemInSortedListOfSeqElem (BLIST ListSortedOfSeqElem,
TIME_SEQUENTIAL_ELEM SeqElem)
{
    int             i;
    TIME_SEQUENTIAL_ELEM AuxSeqElem;

    for (i=0; i < BListSize (ListSortedOfSeqElem); i++)
    {
        AuxSeqElem = (TIME_SEQUENTIAL_ELEM) BListElt (ListSortedOfSeqElem, i);
        if (TimeGetSeqElemSlackAtOutput (SeqElem) <
            TimeGetSeqElemSlackAtOutput (AuxSeqElem))
            break;
    }
    if (BListInsertInList (ListSortedOfSeqElem, SeqElem, i+1))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
/*-----*/
GNL_STATUS TimeSortListOfSeqInstByMinSlackAtOutput (GNL Gnl,
                                                    BLIST ListSortedOfSeqElem)
{
    GNL_VAR          Var;
    int              i, Rank;
    GNL_STATUS       GnlStatus;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT    ComponentI;
    GNL              GnlCompoI;
    GNL_ASSOC        Assoc;
    TIME_SEQUENTIAL_ELEM SeqElem;
    GNL_TIMING_INFO TimingI;
    unsigned int     Key;
    float            Slack;
    BLIST            HierInstPath;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BlistElt (GnlComponents (Gnl), i);
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
            {
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                    return (GNL_MEMORY_FULL);
                if (TimeSortListOfSeqInstByMinSlackAtOutput (GnlCompoI,
                                                ListSortedOfSeqElem))
                    return (GNL_MEMORY_FULL);
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            }
        }
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
            continue;
    }
}
```

timepath.e

```
SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
Assoc = GnlSequentialCompoOutputAssoc (SeqCompo);
if (!Assoc)
    continue;
Var = GnlAssocActualPort (Assoc);
if (!Var)
    continue;
if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
    continue;

if (TimeCreateSeqElem (&SeqElem))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &TimingI, &Key, &Rank))
    return (GNL_MEMORY_FULL);
Slack = GnlTimingInfoRequiredRise (TimingI) -
        GnlTimingInfoArrivalRise (TimingI);
if (Slack > (GnlTimingInfoRequiredRise (TimingI) -
               GnlTimingInfoArrivalRise (TimingI)))
    Slack = GnlTimingInfoRequiredRise (TimingI) -
            GnlTimingInfoArrivalRise (TimingI);
SetTimeGetSeqElemSlackAtOutput (SeqElem, Slack);
SetTimeGetSeqElemSeqComp (SeqElem, SeqCompo);

if (BListCopyNoEltCr (G_PileOfComponent, &HierInstPath))
    return (GNL_MEMORY_FULL);
SetTimeGetSeqElemHierInstPath (SeqElem, HierInstPath);

if (TimeInsertSeqElemInSortedListOfSeqElem (ListSortedOfSeqElem, SeqElem))
    return (GNL_MEMORY_FULL);

}
return (GNL_OK);
}
/*-----*/
GNL_STATUS TimeExpandSequentialCritPathFromGnl (GNL Gnl, LIBC_LIB Lib,
                                                BLIST Path, int MaxPaths)
{
    BLIST                                ListSortedOfSeqElem;
    GNL_SEQUENTIAL_COMPONENT      SeqCompo;
    GNL_ASSOC                         Assoc;
    TIME_SEQUENTIAL_ELEM             SeqElem;
    int                               i;

    if (BListCreate (&ListSortedOfSeqElem))
        return (GNL_MEMORY_FULL);

    if (TimeSortListOfSeqInstByMinSlackAtOutput (Gnl, ListSortedOfSeqElem))
        return (GNL_MEMORY_FULL);

    if (!BListSize (ListSortedOfSeqElem))
    {
        BListQuickDelete (&ListSortedOfSeqElem);
        return (GNL_OK);
    }
```

timepath.e

```
for (i=0; i < BListSize (ListSortedOfSeqElem); i++)
{
    SeqElem = (TIME_SEQUENTIAL_ELEM) BListElt (ListSortedOfSeqElem, i);
    SeqCompo = TimeGetSeqElemSeqComp (SeqElem);
    Assoc = GnlSequentialCompoOutputAssoc (SeqCompo);
    G_PileOfComponent = TimeGetSeqElemHierInstPath (SeqElem);
    if (TimeExpandCritPathFromOutputOfSeqCell (Path, MaxPaths, Assoc, Lib))
        return (GNL_MEMORY_FULL);
    if (TimeConditionsAreSatisfied (MaxPaths))
        break;
    Assoc = GnlSequentialCompoOutputBarAssoc (SeqCompo);
    G_PileOfComponent = TimeGetSeqElemHierInstPath (SeqElem);
    if (TimeExpandCritPathFromOutputOfSeqCell (Path, MaxPaths, Assoc, Lib))
        return (GNL_MEMORY_FULL);
    if (TimeConditionsAreSatisfied (MaxPaths))
        break;
}
BListQuickDelete (&ListSortedOfSeqElem);
return (GNL_OK);
}
/*-----*/
/*-----*/
/* TimeExpandClockFreqForOneClock */
/*-----*/
void TimePrintClockFreqForOneClock (GNL_CLOCK Clock, BLIST ListOfCritPaths)
{
    GNL_PATH_ELEM PathElem;
    BLIST AuxList, Path;
    int i;
    float ClockFreqValue, SetupTime, ArrTime, ReqTime;

    if (!ListOfCritPaths || !Clock)
        return;
    ClockFreqValue = 0.0;
    fprintf (stderr, " -----\\n");
    fprintf (stderr, " Clock : ");
    if (GnlClockHierNameOfGnlClok (Clock))
        fprintf (stderr, "%s/", GnlClockHierNameOfGnlClok (Clock));
    fprintf (stderr, "%s\\n", GnlVarName (GnlClockSourceClock (Clock)));
    fprintf (stderr, " -----\\n");
    for (i=0; i < BListSize (ListOfCritPaths); i++)
    {
        AuxList = (BLIST) BListElt (ListOfCritPaths, i);
        TimePrintPath (AuxList);
        if (!i)
        {
            PathElem = (GNL_PATH_ELEM) BListElt (AuxList, BListSize (AuxList) - 1);
            SetupTime = GnlMaxFloat (GnlPathElemRiseIncr (PathElem),
                                     GnlPathElemFallIncr (PathElem));
            ArrTime = GnlMaxFloat (GnlPathElemRiseArrTime (PathElem),
                                  GnlPathElemFallArrTime (PathElem));
            if (GnlPathElemRiseArrTime (PathElem) >
                GnlPathElemFallArrTime (PathElem))
                ClockFreqValue = GnlPathElemRiseArrTime (PathElem) +
```

timepath.e

```
        GnlPathElemRiseIncr (PathElem);
    else
        ClockFreqValue = GnlPathElemFallArrTime (PathElem) +
                        GnlPathElemFallIncr (PathElem);
    }
    BListDelete (&AuxList, TimeFreePathElem);
}
BListQuickDelete (&ListOfCritPaths);
if (ClockFreqValue)
{
    SetGnlClockArrTime (Clock, ArrTime);
    SetGnlClockSetupTime (Clock, SetupTime);
    fprintf (stderr, " Clock : ");
    if (GnlClockHierNameOfGnlClock (Clock))
        fprintf (stderr,"%s/", GnlClockHierNameOfGnlClock (Clock));
    fprintf (stderr,"%s\n", GnlVarName (GnlClockSourceClock (Clock)));
    fprintf (stderr, " Clock Period \t\t%.2f ns\n", ClockFreqValue);
    fprintf (stderr, " Clock Frequency \t\t%.2f Mhz\n",
            1000 * 1/ClockFreqValue);
    fprintf (stderr, " -----");
    -----"\n");
}
BListQuickDelete (&ListOfCritPaths);
}

/*-----*/
/* TimePrintAllCriticalPath */
/*-----*/
void TimePrintAllCriticalPath (GNL TopGnl, LIBC_LIB Lib,
                               float *MinSlackForCombPath, float *MinSlackForSeqPath)
{
    BLIST          ListOfCritPaths, AuxList;
    float          ClockFreqValue, CombPathValue;
    float          MinSlack;
    GNL_CLOCK     Clock, ClockI, ClockJ;
    int           i, j;
    GNL_PATH_ELEM PathElem;

    GnlPrintHeadOfTimingReport (TopGnl, Lib);
    ClockFreqValue = CombPathValue = 0.0;
    *MinSlackForCombPath = 0.0;
    if (BListElt (MatrixCriticalSection, 0))
    {
        ListOfCritPaths = (BLIST) BListElt (
            (BLIST) BListElt (MatrixCriticalSection, 0), 0);
        for (i=0; i < BListSize (ListOfCritPaths); i++)
        {
            AuxList = (BLIST) BListElt (ListOfCritPaths, i);
            TimePrintPath (AuxList);
            if (!i)
            {
                PathElem = (GNL_PATH_ELEM) BListElt (AuxList, BListSize (AuxList) -
1);
                CombPathValue = GnlMaxFloat (GnlPathElemRiseArrTime (PathElem),
                                              GnlPathElemFallArrTime (PathElem));
                *MinSlackForCombPath = -CombPathValue;
            }
        }
    }
}
```

timepath.e

```
    BListDelete (&AuxList, TimeFreePathElem);
}
BListQuickDelete (&ListOfCritPaths);
}

if (CombPathValue)
{
    fprintf (stderr, " -----\n");
    fprintf (stderr, " Combinational Path \t\t%.3f ns\n", CombPathValue);
    fprintf (stderr, " -----");
    fprintf (stderr, "\n\n");
}

*MinSlackForSeqPath = MAX_FLOAT;
for (i=0; i < BListSize (LocalListOfClocks); i++)
{
    Clock = (GNL_CLOCK) BListElt (LocalListOfClocks, i);
    ListOfCritPaths = (BLIST) BlistElt (
        (BLIST) BListElt (MatrixCriticalPath, i+1), i+1);
    (void) TimePrintClockFreqForOneClock (Clock, ListOfCritPaths);
    MinSlack = GnlClockReqTime (Clock) - GnlClockArrTime (Clock);
    if (*MinSlackForSeqPath > MinSlack)
        *MinSlackForSeqPath = MinSlack;
}
if (!LocalListOfClocks)
    *MinSlackForSeqPath = 0.0;

if (BListSize (LocalListOfClocks) > 1)
{
    fprintf (stderr, "\n\n -----");
    fprintf (stderr, " MIXED CLOCK DOMAIN :\n");
    fprintf (stderr, " -----");
    for (i=0; i < BListSize (LocalListOfClocks); i++)
    {
        ClockI = (GNL_CLOCK) BListElt (LocalListOfClocks, i);
        for (j=0; j < BListSize (LocalListOfClocks); j++)
        {
            ListOfCritPaths = (BLIST) BlistElt (
                (BLIST) BListElt (MatrixCriticalPath, i+1), j+1);
            if ((i == j) || !ListOfCritPaths)
                continue;
            ClockJ = (GNL_CLOCK) BlistElt (LocalListOfClocks, j);
            fprintf (stderr, " The most critical path ");
            if (GnlClockHierNameOfGnlClok (ClockI))
                fprintf (stderr, "%s/", GnlClockHierNameOfGnlClok (ClockI));
            fprintf (stderr, "%s ---> ", GnlVarName (GnlClockSourceClock
(ClockI)));
            if (GnlClockHierNameOfGnlClok (ClockJ))
                fprintf (stderr, "%s/", GnlClockHierNameOfGnlClok (ClockJ));
            fprintf (stderr, "%s\n", GnlVarName (GnlClockSourceClock (ClockJ)));
        }
    }
    AuxList = (BLIST) BListElt (ListOfCritPaths, 0);
    TimePrintPath (AuxList);
```

timepath.e

```
        }
    }

}

/*
 *-----*
 * TimeExpandCritPathFromNetwork
 *-----*/
/* Doc procedure TimeExpandCritPathFromNetwork :
 - Getting combinational critical path and Clock periode from a given
 Network (GNL_NETWORK).
*/
/*
 *-----*
GNL_STATUS TimeExpandCritPathFromNetwork (GNL_NETWORK Nw,
                                             LIBC_LIB     Lib,
                                             int          MaxPaths,
                                             BLIST        ListOfClocks,
                                             double       AreaWithoutNet,
                                             float        MinSlackForCombPath,
                                             float        MinSlackForSeqPath,
                                             int          PrintCritRegion)
{
    GNL                      TopGnl;
    GNL_STATUS    GnlStatus;
    BLIST         Path, AuxList;
    int           i, j;
    GNL_CLOCK     Clock;

    TopGnl = GnlNetworkTopGnl (Nw);

    LocalMinSlackForCombPath = MinSlackForCombPath;
    LocalMinSlackForSeqPath = MinSlackForSeqPath;

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    LocalListOfClocks = ListOfClocks;
    if (BListCreateWithSize(BListSize (LocalListOfClocks)+1,
&MatrixCriticalPath))
        return (GNL_MEMORY_FULL);

    for (i=0; i < BListSize (LocalListOfClocks)+1; i++)
    {
        if (BListCreateWithSize (BListSize (LocalListOfClocks) +1, &AuxList))
            return (GNL_MEMORY_FULL);
        for (j=0; j < BListSize (LocalListOfClocks)+1; j++)
            if (BListAddElt (AuxList, (int) NULL))
                return (GNL_MEMORY_FULL);
        if (BListAddElt (MatrixCriticalPath, (int) AuxList))
            return (GNL_MEMORY_FULL);
    }

    if (MaxPaths == 1 && BListSize (LocalListOfClocks) <= 1)
    {
        LocalIsAboutCombinational = 1;
/*        fprintf (stderr,"Past 1, LocalMinSlackForCombPath = %.2f\n",
LocalMinSlackForCombPath); */
    }
}
```

timepath.e

```
if (LocalMinSlackForCombPath != 0)
if (TimeExpandCombinationalCritPathFromGnl (TopGnl, Lib, MaxPaths))
    return (GNL_MEMORY_FULL);
LocalIsAboutCombinational = 0;

if (BListCreate (&Path))
    return (GNL_MEMORY_FULL);

/*   fprintf (stderr,"Past 2\n"); */
if (!TimeConditionsAreSatisfied (MaxPaths))
    if (TimeExpandCombinationalCritPathFromGnl (TopGnl, Lib, MaxPaths))
        return (GNL_MEMORY_FULL);

/*   fprintf (stderr,"Past 3\n"); */
if (!TimeConditionsAreSatisfied (MaxPaths))
    if (TimeExpandSequentialCritPathFromGnl (TopGnl, Lib, Path, MaxPaths))
        return (GNL_MEMORY_FULL);

BListQuickDelete (&Path);
}
else
{
    if (TimeExpandCombinationalCritPathFromGnl (TopGnl, Lib, MaxPaths))
        return (GNL_MEMORY_FULL);
    if (BListCreate (&Path))
        return (GNL_MEMORY_FULL);

    if (!TimeConditionsAreSatisfied (MaxPaths))
        if (TimeExpandSequentialCritPathFromGnl (TopGnl, Lib, Path, MaxPaths))
            return (GNL_MEMORY_FULL);
    BListQuickDelete (&Path);
}

TimePrintAllCriticalPath(TopGnl, Lib, &MinSlackForCombPath, &MinSlackForSeqPath)
;
if (PrintCritRegion)
    if (GnlStatus = GnlGetEpsiCriticInstancesFromGnl(TopGnl, Lib,
        MinSlackForSeqPath, MinSlackForCombPath, AreaWithoutNet))
        return (GnlStatus);

BListQuickDelete (&G_PileOfComponent);
return (GNL_OK);
}
/* ----- EOF ----- */
```

timepath.e

```
/*-----*/
/*
 *      File:          timepath.e
 *      Version:       1.1
 *      Modifications: -
 *      Documentation: -
 */
/*
 *      Description: *
 */
/*-----*/
GNL_STATUS TimeGetCritPathFromVar (BLIST Path, int NbCp, GNL_VAR Var,
                                   LIBC_LIB Lib, float RArrTime, float FArrTime,
                                   float InTransRise, float InTransFall,int Start);

GNL_STATUS TimeExpandCritPathFromNetwork (GNL_NETWORK Nw,
                                         LIBC_LIB     Lib,
                                         int          MaxPaths,
                                         BLIST        ListOfClocks,
                                         double       AreaWithoutNet,
                                         float        MinSlackForCombPath,
                                         float        MinSlackForSeqPath,
                                         int          PrintCritRegion);

/*-----*/
/*-----EOF-----*/
/*-----*/
```

timeutil.c

timeutil.c

```
SetGnlTimingInfoHook (*TimingInfo, NULL);

    return (GNL_OK);
}

/*-----*/
/* GnlFreeTimingInfo */
/*-----*/
void GnlFreeTimingInfo (GNL_TIMING_INFO *TimingInfo)
{
    if (!(*TimingInfo))
        return;

    if (GnlTimingInfoHook(*TimingInfo))
        GnlFreeTimingInfo ((GNL_TIMING_INFO *) &GnlTimingInfoHook(*TimingInfo));

    free ((char *) *TimingInfo);
    (*TimingInfo) = NULL;
}

/*-----*/
/*-----*/
/* TimeCopyTimingInfo */
/*-----*/
GNL_STATUS TimeCopyTimingInfo (GNL_TIMING_INFO TimingI1,
                               GNL_TIMING_INFO *TimingI2)
{
    GNL_TIMING_INFO AuxTimingI;

    if (!TimingI1)
    {
        *TimingI2 = NULL;
        return (GNL_OK);
    }

    if (GnlCreateTimingInfo (TimingI2))
        return (GNL_MEMORY_FULL);

    SetGnlTimingInfoArrivalFall (*TimingI2, GnlTimingInfoArrivalFall
(TimingI1));
    SetGnlTimingInfoArrivalRise (*TimingI2, GnlTimingInfoArrivalRise
(TimingI1));
    SetGnlTimingInfoRequiredFall
(*TimingI2,GnlTimingInfoRequiredFall(TimingI1));
    SetGnlTimingInfoRequiredRise
(*TimingI2,GnlTimingInfoRequiredRise(TimingI1));
    SetGnlTimingInfoFanout (*TimingI2, GnlTimingInfoFanout (TimingI1));
    SetGnlTimingInfoCapa (*TimingI2, GnlTimingInfoCapa (TimingI1));
    SetGnlTimingInfoTransitionFall (*TimingI2,
                                    GnlTimingInfoTransitionFall (TimingI1));
    SetGnlTimingInfoTransitionRise (*TimingI2,
                                    GnlTimingInfoTransitionRise(TimingI1));
    SetGnlTimingInfoTag (*TimingI2, GnlTimingInfoTag (TimingI1));
    if (GnlTimingInfoHook (TimingI1))
    {
        TimeCopyTimingInfo ((GNL_TIMING_INFO) GnlTimingInfoHook (TimingI1),
```

timeutil.c

```
        &AuxTimingI);
    SetGnlTimingInfoHook (*TimingI2, AuxTimingI);
}

return (GNL_OK);
}

/*-----
/* TimeCreateSeqElem
/*-----*/
GNL_STATUS TimeCreateSeqElem (TIME_SEQUENTIAL_ELEM *SeqElem)
{
    if ((*SeqElem = (TIME_SEQUENTIAL_ELEM)
          calloc (1, sizeof(TIME_SEQUENTIAL_ELEM_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetTimeGetSeqElemSeqComp (*SeqElem, NULL);
    SetTimeGetSeqElemHierInstPath (*SeqElem, NULL);
    SetTimeGetSeqElemSlackAtOutput (*SeqElem, 0.0);

    return (GNL_OK);
}

/*-----
/* GnlCreateCritPath
/*-----*/
GNL_STATUS GnlCreateCritPath (GNL_CRITICAL_PATH *CritPath)
{
    if ((*CritPath = (GNL_CRITICAL_PATH)
          calloc (1, sizeof(GNL_CRITICAL_PATH_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlCritPathCriticalVar (*CritPath, NULL);
    SetGnlCritPathCriticalAssoc (*CritPath, NULL);
    SetGnlCritPathIncr (*CritPath, 0.0);
    SetGnlCritPathInstName (*CritPath, NULL);
    SetGnlCritPathArrTime (*CritPath, -MAX_FLOAT);
    SetGnlCritPathReqTime (*CritPath, MAX_FLOAT);
    SetGnlCritPathPredecessors (*CritPath, NULL);
    SetGnlCritPathSuccessors (*CritPath, NULL);

    return (GNL_OK);
}

/*-----
/* GnlAddPredecessorToCritPath
/*-----*/
GNL_STATUS GnlAddPredecessorToCritPath (GNL_CRITICAL_PATH CritPath,
                                         GNL_CRITICAL_PATH Predecessor)
{
    BLIST ListPredecessors;

    if (!CritPath || !Predecessor)
        return (GNL_OK);

    if (!GnlCritPathPredecessors (CritPath))
    {
```

timeutil.c

```
if (BListCreateWithSize (1, &ListPredecessors))
    return (GNL_MEMORY_FULL);
SetGnlCritPathPredecessors (CritPath, ListPredecessors);
}

ListPredecessors = GnlCritPathPredecessors (CritPath);

if (BListAddElt (ListPredecessors, Predecessor))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* GnlAddSuccessorToCritPath
*/
-----
GNL_STATUS GnlAddSuccessorToCritPath (GNL_CRITICAL_PATH CritPath,
                                      GNL_CRITICAL_PATH Successor)
{
    BLIST      ListSuccessors;

    if (!CritPath || !Successor)
        return (GNL_OK);

    if (!GnlCritPathSuccessors (CritPath))
    {
        if (BListCreateWithSize (1, &ListSuccessors))
            return (GNL_MEMORY_FULL);
        SetGnlCritPathSuccessors (CritPath, ListSuccessors);
    }

    ListSuccessors = GnlCritPathSuccessors (CritPath);

    if (BListAddElt (ListSuccessors, Successor))
        return (GNL_MEMORY_FULL);

    return (GNL_OK);
}

/*-----
/* GnlFreeCritPath
*/
-----
void GnlFreeCritPath (GNL_CRITICAL_PATH *CritPath)
{
    GNL_CRITICAL_PATH      Succesor;
    int                  i, Rank;

    if (!(*CritPath))
        return;

    if (GnlCritPathPredecessors (*CritPath))
        BListDelete (&GnlCritPathPredecessors (*CritPath), GnlFreeCritPath);

    for (i=0; i < BListSize (GnlCritPathSuccessors (*CritPath)); i++)
    {
        Succesor = (GNL_CRITICAL_PATH) BListElt (GnlCritPathSuccessors
(*CritPath),
```

```

        i);
    Rank = BListMemberOfList (GnlCritPathPredecessors (Succesor), *CritPath,
                             IntIdentical);
    BListDelShift (GnlCritPathPredecessors (Succesor), Rank);
}
BListQuickDelete (&GnlCritPathSuccessors (*CritPath));

if (GnlCritPathInstName (*CritPath))
    free (GnlCritPathInstName (*CritPath));

free ((char *) *CritPath);
(*CritPath) = NULL;
}

/*-----*/
/* GnlFreeTimingInfoFromVar */
/*-----*/
/* Doc procedure GnlFreeTimingInfoFromVar :
   - For a given Net or Var (GNL_VAR) :
     - Free the timing info attached.
*/
/*-----*/
GNL_STATUS GnlFreeTimingInfoFromVar (GNL_VAR Var, int IsInTheTopGnl)
{
    int          i;
    BLIST        SubList;

    if (IsInTheTopGnl)
        GnlFreeTimingInfo ((GNL_TIMING_INFO *) &GnlVarTraversalInfoHook (Var));
    else
    {
        for (i=0; i<BListSize ((BLIST)GnlVarTraversalInfoHook (Var)); i++)
        {
            SubList = (BLIST)BListElt ((BLIST)GnlVarTraversalInfoHook (Var),
                                         i);
            if (!SubList)
                continue;

            BListDelete (&SubList, GnlFreeTimingInfo);
        }
        BListQuickDelete ((BLIST *) &GnlVarTraversalInfoHook (Var));
    }

    return (GNL_OK);
}

/*-----*/
/* GnlFreeTimingInfoFromGnl */
/*-----*/
/* Doc procedure GnlFreeTimingInfoFromGnl :
   - For a given Gnl (GNL) :
     - Free Hierarchically The timing info attached to each Net "GNLVAR".
*/
/*-----*/
GNL_STATUS GnlFreeTimingInfoFromGnl (GNL Gnl, int IsInTheTopGnl)
{

```

timeutil.c

```
GNL_VAR          Var;
int             i, j;
GNL_STATUS      GnlStatus;
GNL             GnlCompoI;
GNL_COMPONENT   ComponentI;
BLIST           Components, BucketI;

Components = GnlComponents (Gnl);
for (i=0; i<BListSize (Components); i++)
{
    ComponentI = (GNL_COMPONENT) BListElt (Components, i);
    if (GnlComponentType (ComponentI) != GNL_USER_COMPO)
        continue;
    GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
    if (GnlCompoI)
    {
        if (GnlStatus = GnlFreeTimingInfoFromGnl (GnlCompoI, 0))
            return (GnlStatus);
    }
}

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;
        if ((GnlStatus = GnlFreeTimingInfoFromVar (Var, IsInTheTopGnl)))
            return (GnlStatus);
    }
}

return (GNL_OK);
}

/*-----*/
/* GnlFreeTimingInfoFromNetwork */
/*-----*/
/* Doc procedure GnlFreeTimingInfoFromNetwork :
 - For a given Network (GNL_NETWORK) :
     - Free Hierarchically The timing info attached to each Net "GNLVAR".
*/
/*-----*/
GNL_STATUS GnlFreeTimingInfoFromNetwork (GNL_NETWORK Nw)
{
    GNL           TopGnl;
    GNL_STATUS    GnlStatus;

    TopGnl = GnlNetworkTopGnl (Nw);

    if ((GnlStatus = GnlFreeTimingInfoFromGnl (TopGnl, 1)))
        return (GnlStatus);

    return (GNL_OK);
}
```

```

/*-----*/
int GnlCritPath_InstNameCmpStr (GNL_CRITICAL_PATH CritPath, char *Str)
{
    if (!GnlCritPathInstName (CritPath) || !Str)
        return (Str == GnlCritPathInstName (CritPath));

    if (!strcmp (GnlCritPathInstName (CritPath), Str))
        return (1);

    return (0);
}
/*-----*/
int GnlFloatEpsilonEqual (float a, float b, float Epsi)
{
    if (a > b)
    {
        if ((a - b) <= Epsi)
            return (1);
    }
    else
    {
        if ((b - a) <= Epsi)
            return (1);
    }

    return (0);
}

/*-----*/
/* GnlGetEpsiCritInstFromCombCell */
/*-----*/
GNL_STATUS GnlGetEpsiCritInstFromCombCell (GNL_USER_COMPONENT UserCompo,
                                             GNL_ASSOC      Assoc,
                                             LIBC_LIB       Lib,
                                             BLIST          ListCritInst,
                                             float          Epsi,
                                             int *IsEndInput, int *IsEndSeq,
                                             float          MinSlack,
                                             int           IsStartOutput,
                                             int           IsAboutComb)
{
    LIBC_CELL      Cell;
    LIBC_PIN       PinOut, PinIn;
    int           i, Rank, Fanout, IsEndInputAux, IsEndSeqAux;
    GNL_ASSOC     AssocI;
    char          *Formal, *HierarchycalName;
    GNL_VAR       Var, Vari;
    GNL_TIMING_INFO TimingI, Timing;
    float          InTransRise, InTransFall, OutTransR, OutTransF,
                  DelayR, DelayF, TimeRise, TimeFall;
}

```

```

float Length, WireCapa, WireResistance, OutCapa;
float Max, MaxI;
BLIST Interface;
GNL_STATUS GnlStatus;
GNL_CRITICAL_PATH CritPath;
unsigned int Key;
BLIST SubList;

Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinOut) == INPUT_E )
    return (GNL_OK);

if (GnlGetHierarchycalInstName (GnlUserComponentInstName (UserCompo),
                                &HierarchycalName))
    return (GNL_MEMORY_FULL);
Key = KeyOfName (HierarchycalName, BListSize (ListCritInst));
SubList = (BLIST)BListElt (ListCritInst, Key);
if (!SubList)
{
    if (BListCreate (&SubList))
        return (GNL_MEMORY_FULL);
    BListElt (ListCritInst, Key) = (int) SubList;
}

if (BListMemberOfList (SubList,
HierarchycalName, GnlCritPath_InstNameCmpStr))
{
    free (HierarchycalName);
    return (GNL_OK);
}
free (HierarchycalName);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &Timing, &Key, &Rank))
    return (GNL_MEMORY_FULL);

Max = GnlTimingInfoArrivalRise (Timing);
if (Max < GnlTimingInfoArrivalFall (Timing))
    Max = GnlTimingInfoArrivalFall (Timing);
Fanout = GnlTimingInfoFanout (Timing);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (Timing);

Interface = GnlUserComponentInterface (UserCompo);

for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))

```

```

        continue;
Formal = (char *) GnlAssocFormalPort (AssocI);
if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
    continue;
if (LibPinDirection(PinIn) != INPUT_E)
    continue;
if (!LibGetArcTiming (PinIn, PinOut))
    continue;
IsEndInputAux = IsEndSeqAux = 0;
if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                &TimingI, &Key, &Rank)))
    return (GnlStatus);
InTransRise = GnlTimingInfoTransitionRise (TimingI);
InTransFall = GnlTimingInfoTransitionFall (TimingI);

LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                 Fanout, Cell, PinIn, PinOut,
                 &DelayR, &OutTransR,
                 &DelayF, &OutTransF);
GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                DelayR, DelayF,
                                                GnlTimingInfoArrivalRise (TimingI),
                                                GnlTimingInfoArrivalFall (TimingI),
                                                &TimeRise, &TimeFall);
MaxI = TimeRise;
if (MaxI < TimeFall)
    MaxI = TimeFall;
if (!GnlFloatEpsilonEqual (Max, MaxI, Max * Epsi))
    continue;
if (GnlStatus = GnlGetEpsiCritInstFromVar (VarI, Lib, ListCritInst,
Epsi,
                &IsEndInputAux, &IsEndSeqAux, MinSlack,
                1, IsStartOutput, IsAboutComb))
    return (GnlStatus);
if (IsEndInputAux)
    *IsEndInput = 1;
if (IsEndSeqAux)
    *IsEndSeq = 1;
}

if ((GnlStatus = GnlCreateCritPath (&CritPath)))
    return (GnlStatus);
SetGnlCritPathCriticalAssoc (CritPath, Assoc);
if (GnlTimingInfoArrivalRise(Timing) >
GnlTimingInfoArrivalFall(Timing))
{
    SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalRise
(Timing));
    SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredRise
(Timing));
}
else
{
    SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalFall
(Timing));
    SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredFall
(Timing));
}

```

```

        }

    SetGnlCritPathInstNameFromUserCompo (CritPath,
                                         GnlUserComponentInstName (UserCompo));
    if (IsAboutComb)
    {
        if (*IsEndInput)
        {
            if (BListAddElt (SubList, CritPath))
                return (GNL_MEMORY_FULL);
        }
        else
            GnlFreeCritPath (&CritPath);
    }
    else
    {
        if (!IsStartOutput || *IsEndSeq)
        {
            if (BListAddElt (SubList, CritPath))
                return (GNL_MEMORY_FULL);
        }
        else
            GnlFreeCritPath (&CritPath);
    }

    return (GNL_OK);
}
/*-----*/
/* GnlGetEpsiCritInstFromUserCompo */
/*-----*/
GNL_STATUS GnlGetEpsiCritInstFromUserCompo (GNL_ASSOC Assoc,
                                             LIBC_LIB     Lib,
                                             BLIST        ListCritInst,
                                             float        Epsi,
                                             int *IsEndInput, int *IsEndSeq,
                                             float        MinSlack,
                                             int          IsStartOutput,
                                             int          IsAboutComb,
                                             GNL_VAR      Actual)
{
    GNL_VAR           Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS        GnlStatus;
    int               Rank;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);

    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
    {
        /* It corresponds to a LIBC library cell. */
        if (GnlUserComponentCellDef (UserCompo))
        {
            if (GnlStatus = GnlGetEpsiCritInstFromCombCell (UserCompo,
                                                          Assoc, Lib, ListCritInst, Epsi,
                                                          IsEndInput, IsEndSeq, MinSlack,

```

timeutil.c

```
                                IsStartOutput, IsAboutComb))  
    return (GnlStatus);  
  
    return (GNL_OK);  
}  
else  
/* It is a black box without any definition so we stop */  
/* at its boundary. */  
{  
    fprintf (stderr,  
        "# WARNING: black box <%s %s> may modify final estimation\n",  
        GnlUserComponentName (UserCompo),  
        GnlUserComponentInstName (UserCompo));  
}  
return (GNL_OK);  
}  
  
if (BListAddElt (G_PileOfComponent, UserCompo))  
    return (GNL_MEMORY_FULL);  
  
if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,  
&AuxFormal))  
    return (GnlStatus);  
  
if (GnlStatus = GnlGetEpsiCritInstFromVar (AuxFormal, Lib, ListCritInst,  
Epsi,  
        IsEndInput, IsEndSeq, MinSlack,  
        0, IsStartOutput, IsAboutComb))  
    return (GnlStatus);  
BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));  
  
return (GNL_OK);  
}  
/*-----*/  
/* GnlGetEpsiCritInstFromSeqCell */  
/*-----*/  
GNL_STATUS GnlGetEpsiCritInstFromSeqCell (GNL_ASSOC Assoc,  
                                         BLIST ListCritInst,  
                                         float Epsi,  
                                         int *IsEndInput, int *IsEndSeq,  
                                         float MinSlack)  
{  
    LIBC_CELL Cell;  
    LIBC_PIN PinOut;  
    int i, Rank;  
    GNL_ASSOC AssocI;  
    char *Formal, *HierarchycalName;  
    GNL_VAR Var;  
    GNL_TIMING_INFO TimingI;  
    float SlackRise, SlackFall, Slack;  
    GNL_STATUS GnlStatus;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    GNL_CRITICAL_PATH CritPath;  
    unsigned int Key;  
    BLIST SubList;
```

timeutil.c

```
SeqCompo = (GNL_SEQUENTIAL_COMPONENT) GnlAssocTraversalInfoComponent
(Assoc);
Cell = GnlSeqCompoInfoCell (SeqCompo);
Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))
    return (GNL_OK);
if (LibPinDirection (PinOut) != OUTPUT_E )
    return (GNL_OK);

if (GnlGetHierarchicalInstName (GnlSequentialCompoInstName (SeqCompo),
                               &HierarchicalName))
    return (GNL_MEMORY_FULL);
Key = KeyOfName (HierarchicalName, BListSize (ListCritInst));
SubList = (BLIST)BListElt (ListCritInst, Key);
if (!SubList)
{
    if (BListCreate (&SubList))
        return (GNL_MEMORY_FULL);
    BListElt (ListCritInst, Key) = (int) SubList;
}

if (BListMemberOfList (SubList,
HierarchicalName, GnlCritPath_InstNameCmpStr))
{
    free (HierarchicalName);
    return (GNL_OK);
}
free (HierarchicalName);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &TimingI, &Key, &Rank))
    return (GNL_MEMORY_FULL);

if ((GnlStatus = GnlCreateCritPath (&CritPath)))
    return (GnlStatus);
SetGnlCritPathCriticalAssoc (CritPath, Assoc);
if (GnlTimingInfoArrivalFall (TimingI) < GnlTimingInfoArrivalRise
(TimingI))
{
    SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalRise (TimingI));
    SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredRise (TimingI));
}
else
{
    SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalFall (TimingI));
    SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredFall (TimingI));
}

SetGnlCritPathInstNameFromUserCompo (CritPath,
                                      GnlSequentialCompoInstName (SeqCompo));
if (BListAddElt (SubList, CritPath))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}/*-----*/
```

timeutil.c

```
/* GnlGetEpsiCritInstFFrom3State */  
/*-----*/  
GNL_STATUS GnlGetEpsiCritInstFFrom3State (GNL_ASSOC Assoc,  
                                         LIBC_LIB Lib,  
                                         BLIST ListCritInst,  
                                         int Epsi,  
                                         int *IsEndInput, int *IsEndSeq,  
                                         float MinSlack,  
                                         int IsStartOutput,  
                                         int IsAboutComb)  
{  
    LIBC_CELL Cell;  
    LIBC_PIN PinOut, PinIn;  
    int i, Rank, Fanout;  
    GNL_ASSOC AssocI;  
    char *Formal, *HierarchycalName;  
    GNL_VAR Var, VarI;  
    GNL_TIMING_INFO Timing, TimingI;  
    float InTransRise, InTransFall, OutTransR, OutTransF,  
          DelayR, DelayF, TimeRise, TimeFall;  
    float Length, WireCapa, WireResistance, OutCapa;  
    float Max, MaxI;  
    BLIST Interface;  
    GNL_STATUS GnlStatus;  
    GNL_CRITICAL_PATH CritPath;  
    GNL_TRISTATE_COMPONENT TriStateCompo;  
    unsigned int Key;  
    BLIST SubList;  
  
    TriStateCompo = (GNL_TRISTATE_COMPONENT) GnlAssocTraversalInfoComponent  
(Assoc);  
    Cell = GnlTriStateCompoInfoCell (TriStateCompo);  
    Var = GnlAssocActualPort (Assoc);  
    Formal = (char *) GnlAssocFormalPort (Assoc);  
  
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))  
        return (GNL_OK);  
    if (LibPinDirection (PinOut) == INPUT_E )  
        return (GNL_OK);  
  
    if (GnlGetHierarchycalInstName (GnlUserComponentInstName (TriStateCompo),  
                                    &HierarchycalName))  
        return (GNL_MEMORY_FULL);  
  
    Key = KeyOfName (HierarchycalName, BlistSize (ListCritInst));  
    SubList = (BLIST) BListElt (ListCritInst, Key);  
    if (!SubList)  
    {  
        if (BListCreate (&SubList))  
            return (GNL_MEMORY_FULL);  
        BListElt (ListCritInst, Key) = (int) SubList;  
    }  
  
    if (BListMemberOfList (SubList,  
                         HierarchycalName, GnlCritPath_InstNameCmpStr))  
    {
```

```

    free (HierarchycalName);
    return (GNL_OK);
}
free (HierarchycalName);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &Timing, &Key, &Rank))
    return (GNL_MEMORY_FULL);
Max = GnlTimingInfoArrivalRise (Timing);
if (Max < GnlTimingInfoArrivalFall (Timing))
    Max = GnlTimingInfoArrivalFall (Timing);
Fanout = GnlTimingInfoFanout (Timing);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (Timing);

Interface = GnlTriStateInterface (TriStateCompo);
for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    if (!AssocI)
        continue;
    VarI = GnlAssocActualPort (AssocI);
    if (!VarI)
        continue;
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinIn) != INPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
        &TimingI, &Key, &Rank)))
        InTransRise = GnlTimingInfoTransitionRise (TimingI);
        InTransFall = GnlTimingInfoTransitionFall (TimingI);

        LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
            Fanout, Cell, PinIn, PinOut,
            &DelayR, &OutTransR,
            &DelayF, &OutTransF);

    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
        DelayR, DelayF,
        GnlTimingInfoArrivalRise (TimingI),
        GnlTimingInfoArrivalFall (TimingI),
        &TimeRise, &TimeFall);

    MaxI = TimeRise;
    if (MaxI < TimeFall)
        MaxI = TimeFall;
    if (!GnlFloatEpsilonEqual (Max, MaxI, Max * Epsi))
        continue;
}

```

timeutil.c

```
    if (GnlStatus = GnlGetEpsiCritInstFromVar (VarI, Lib,
ListCritInst,Epsi,
                    IsEndInput, IsEndSeq, MinSlack,
                    1, IsStartOutput, IsAboutComb))
        return (GnlStatus);
    }

    if ((GnlStatus = GnlCreateCritPath (&CritPath)))
        return (GnlStatus);
    SetGnlCritPathCriticalAssoc (CritPath, Assoc);
    if (GnlTimingInfoArrivalRise(Timing) >
GnlTimingInfoArrivalFall(Timing))
    {
        SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalRise (Timing));
        SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredRise
(Timing));
    }
    else
    {
        SetGnlCritPathArrTime (CritPath, GnlTimingInfoArrivalFall (Timing));
        SetGnlCritPathReqTime (CritPath, GnlTimingInfoRequiredFall
(Timing));
    }

    SetGnlCritPathInstNameFromUserCompo (CritPath,
                                         GnlUserComponentInstName (TriStateCompo));
    if (IsAboutComb)
    {
        if (*IsEndInput)
        {
            if (BListAddElt (SubList, CritPath))
                return (GNL_MEMORY_FULL);
        }
        else
            GnlFreeCritPath (&CritPath);
    }
    else
    {
        if (!IsStartOutput || *IsEndSeq)
        {
            if (BListAddElt (SubList, CritPath))
                return (GNL_MEMORY_FULL);
        }
        else
            GnlFreeCritPath (&CritPath);
    }

    return (GNL_OK);
}
/*-----*/
/* GnlGetEpsiCritInstFromAssoc */
/*-----*/
GNL_STATUS GnlGetEpsiCritInstFromAssoc (GNL_ASSOC      Assoc,
                                         LIBC_LIB      Lib,
                                         BLIST        ListCritInst,
                                         float         Epsi,
                                         int *IsEndInput, int *IsEndSeq,
```

```

        float      MinSlack,
        int       IsStartOutput,
        int       IsAboutComb,
        GNL_VAR          Actual)
{
    GNL_STATUS      GnlStatus;
    int           Rank;
    GNL_COMPONENT   GnlCompo;

GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

switch (GnlComponentType (GnlCompo)) {

    case GNL_SEQUENTIAL_COMPO:
        if (!IsAboutComb)
            if (GnlStatus = GnlGetEpsiCritInstFromSeqCell (Assoc,
                    ListCritInst, Epsi, IsEndInput,
                    IsEndSeq, MinSlack))
                return (GnlStatus);
        *IsEndSeq = 1;
        break;

    case GNL_USER_COMPO:
        if (GnlStatus = GnlGetEpsiCritInstFromUserCompo (Assoc, Lib,
                    ListCritInst, Epsi, IsEndInput,
                    IsEndSeq, MinSlack, IsStartOutput,
                    IsAboutComb, Actual))
            return (GnlStatus);
        break;

    case GNL_TRISTATE_COMPO:
        if (GnlStatus = GnlGetEpsiCritInstFFrom3State (Assoc, Lib,
                    ListCritInst, Epsi, IsEndInput,
                    IsEndSeq, MinSlack,
                    IsStartOutput, IsAboutComb))
            return (GnlStatus);
        break;

    case GNL_MACRO_COMPO:
        break;

    default:
        GnlError (12 /* Unknown component */);
        break;
    }
return (GNL_OK);
}

/*-----*/
/* GnlGetEpsiCritInstFromVar */
/*-----*/
GNL_STATUS GnlGetEpsiCritInstFromVar (GNL_VAR Var,
        LIBC_LIB     Lib,
        BLIST        ListCritInst,
        float         Epsi,
        int *IsEndInput, int *IsEndSeq,
        float         MinSlack,

```

```

int           InoutIsIn,
int           IsStartOutput,
int           IsAboutComb)
{
    GNL_VAR          VarI, Actual;
    GNL_ASSOC        AuxAssocVar;
    int              i, Rank;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO Timing, TimingI;
    float            ArrivalTime, RequiredTime,
                     SlackRise, SlackFall, Slack, Max, MaxI;
    GNL_USER_COMPONENT InstOfGnl;
    BLIST            AssocSources, RightVarAssigneds;
    unsigned int      Key;

    if ((GnlStatus = GnlTimingInfoCorrespsToCurrentPathFromVar (Var,
                                                               &Timing, &Key, &Rank)))
        return (GnlStatus);

    Max = GnlTimingInfoArrivalRise (Timing);
    if (Max < GnlTimingInfoArrivalFall (Timing))
        Max = GnlTimingInfoArrivalFall (Timing);

    if (GnlVarDir (Var) == GNL_VAR_INPUT ||
        (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn) )
    {
        if (!BListSize (G_PileOfComponent))
        {
            *IsEndInput = 1;
            return (GNL_OK);
        }
        else
        {
            InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                              BListSize (G_PileOfComponent)-1);

            BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var,
                                                               InstOfGnl, &Actual);
            if (!AuxAssocVar)
            {
                if (BListAddElt (G_PileOfComponent, InstOfGnl))
                    return (GNL_MEMORY_FULL);
                return (GNL_OK);
            }
            if (GnlGetEpsiCritInstFromVar (Actual, Lib, ListCritInst, Epsi,
                                           IsEndInput, IsEndSeq, MinSlack,
                                           1, IsStartOutput, IsAboutComb))
                return (GNL_MEMORY_FULL);

            if (BListAddElt (G_PileOfComponent, InstOfGnl))
                return (GNL_MEMORY_FULL);
        }
    }

    if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))

```

```

    return (GNL_MEMORY_FULL);

    for (i=0; i < BListSize (RightVarAssigneds); i++)
    {
        VarI = (GNL_VAR) BListElt (RightVarAssigneds, i);

        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingI, &Key, &Rank)))
            return (GnlStatus);
        MaxI = GnlTimingInfoArrivalRise (TimingI);
        if (MaxI < GnlTimingInfoArrivalFall (TimingI))
            MaxI = GnlTimingInfoArrivalFall (TimingI);
        if ( GnlFloatEpsilonEqual (Max, MaxI, Max * Epsi) )
            if ((GnlStatus = GnlGetEpsiCritInstFromVar (VarI, Lib, ListCritInst,
                                                          Epsi, IsEndInput, IsEndSeq, MinSlack,
                                                          1, IsStartOutput, IsAboutComb)))
                return (GnlStatus);
    }

    BListQuickDelete (&RightVarAssigneds);

    for (i=0; i < BListSize (AssocSources); i++)
    {
        AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
        if ((GnlStatus = GnlGetEpsiCritInstFromAssoc (AuxAssocVar,
                                                       Lib, ListCritInst,
                                                       Epsi, IsEndInput, IsEndSeq, MinSlack,
                                                       IsStartOutput, IsAboutComb, Var)))
            return (GnlStatus);
    }

    BListQuickDelete (&AssocSources);

    return (GNL_OK);
}

/*-----
/* GnlPrintCriticalInst
/*-----
/* Doc procedure GnlPrintCriticalInst :
   - Priniting the list of epsilon-critical instances.
*/
/*-----
char G_CriticalInstName[528];
GnlPrintCriticalInst (BLIST ListCritInst, double *AreaCritRegion)
{
    int          i, j;
    GNL_CRITICAL_PATH      CritPath;
    GNL_ASSOC      Assoc;
    GNL_VAR       Var, Formal;
    GNL_COMPONENT  Compo;
    LIBC_CELL     Cell;
    BLIST         SubList;

    if (!ListCritInst || ! BListSize (ListCritInst))
        return;
}

```

timeutil.c

timeutil.c

```
    return (GNL_MEMORY_FULL);
BSize (ListCritInst) = HASH_LIST_PATH_ELEM;

AuxEpsi = Epsi;
if (MinCombSlack < 0.0)
    AuxEpsi = Epsi * (-1);
for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
                      (GnlVarDir (Var) != GNL_VAR_INOUT)) )
            continue;

        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
            return (GnlStatus);

        IsEndInput = IsEndSeq = 0;
        Slack = GnlTimingInfoRequiredRise (TimingInfo) -
                GnlTimingInfoArrivalRise (TimingInfo);

        if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo))
            Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                    GnlTimingInfoArrivalFall (TimingInfo);
        if (!GnlFloatEpsilonEqual (MinCombSlack, Slack, MinCombSlack * AuxEpsi))
            continue;

        if (GnlGetEpsiCritInstFromVar (Var, Lib, ListCritInst, Epsi,
                                       &IsEndInput, &IsEndSeq, MinCombSlack, 0, 1, 1))
            return (GNL_MEMORY_FULL);
    }
}

GnlPrintCriticalInst (ListCritInst, AreaCritRegion);

for (i=0; i < BListSize (ListCritInst); i++)
{
    if (!BListElt (ListCritInst, i))
        continue;
    BListDelete ((BLIST*) &BListElt (ListCritInst, i), GnlFreeCritPath);
}
BListQuickDelete (&ListCritInst);

return (GNL_OK);
}

/*-----*/
/*-----*/
/* GnlGetSeqEpsiCriticInstancesFromFFInGnl */
/*-----*/
/*-----*/
/* Doc procedure GnlGetSeqEpsiCriticInstancesFromFFInGnl :
 - For a given Gnl (GNL) :
```

timeutil.c

```
- Get The sequential critical region starting from the inputs of Flops.  
*/  
/*-----*/  
GNL_STATUS GnlGetSeqEpsiCriticInstancesFromFFInGnl (GNL Gnl, LIBC_LIB Lib,  
                                                    float MinSeqSlack,  
                                                    BLIST ListCritInst, float Epsi)  
{  
    GNL_VAR Var;  
    int i, Rank, IsEndInput, IsEndSeq;  
    GNL_STATUS GnlStatus;  
    GNL_TIMING_INFO TimingInfo;  
    float Slack;  
    GNL_SEQUENTIAL_COMPONENT SeqCompo;  
    GNL_COMPONENT ComponentI;  
    GNL GnlCompoI;  
    GNL_ASSOC Assoc;  
    unsigned int Key;  
    float AuxEpsi;  
  
    AuxEpsi = Epsi;  
    if (MinSeqSlack < 0.0)  
        AuxEpsi = Epsi * (-1);  
  
    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)  
    {  
        ComponentI = (GNL_COMPONENT) BlistElt (GnlComponents (Gnl), i);  
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)  
        {  
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);  
            if (GnlCompoI)  
            {  
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))  
                    return (GNL_MEMORY_FULL);  
                if (GnlStatus = GnlGetSeqEpsiCriticInstancesFromFFInGnl (GnlCompoI,  
Lib,  
                                                MinSeqSlack, ListCritInst, Epsi))  
                    return (GnlStatus);  
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));  
            }  
        }  
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)  
            continue;  
  
        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;  
        Assoc = GnlSequentialCompoInputAssoc (SeqCompo);  
        if (!Assoc)  
            continue;  
        Var = GnlAssocActualPort (Assoc);  
        if (!Var)  
            continue;  
        if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))  
            continue;  
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,  
                                                               &TimingInfo, &Key, &Rank)))  
            return (GnlStatus);  
  
        Slack = GnlTimingInfoRequiredRise (TimingInfo) -
```

```

        GnlTimingInfoArrivalRise (TimingInfo) ;
IsEndInput = IsEndSeq = 0;
Slack = GnlTimingInfoRequiredRise (TimingInfo) -
        GnlTimingInfoArrivalRise (TimingInfo) ;
if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -
        GnlTimingInfoArrivalFall (TimingInfo))
    Slack = GnlTimingInfoRequiredFall (TimingInfo) -
        GnlTimingInfoArrivalFall (TimingInfo) ;
if (!GnlFloatEpsilonEqual (MinSeqSlack, Slack, MinSeqSlack * AuxEpsi))
    continue;

    if (GnlGetEpsiCritInstFromVar (Var, Lib, ListCritInst, Epsi,
                                   &IsEndInput, &IsEndSeq, MinSeqSlack, 1, 0, 0))
        return (GNL_MEMORY_FULL);
}

return (GNL_OK);
}
/*-----*/
/* GnlGetSeqEpsiCriticInstancesFromGnl */
/*-----*/
/* Doc procedure GnlGetSeqEpsiCriticInstancesFromGnl :
   - For a given Gnl (GNL) :
     - Get The sequential critical region.
*/
/*-----*/
GNL_STATUS GnlGetSeqEpsiCriticInstancesFromGnl (GNL Gnl, LIBC_LIB Lib,
                                                float MinSeqSlack, float Epsi,
                                                double *AreaCritRegion)
{
    GNL_VAR Var;
    int i, j, Rank, IsEndInput, IsEndSeq;
    GNL_STATUS GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float MinSlack, Slack;
    BLIST BucketI, ListCritInst;
    unsigned int Key;
    float AuxEpsi;

    if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &ListCritInst))
        return (GNL_MEMORY_FULL);
    BSize (ListCritInst) = HASH_LIST_PATH_ELEM;

    AuxEpsi = Epsi;
    if (MinSeqSlack < 0.0)
        AuxEpsi = Epsi * (-1);

    for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
    {
        BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
        for (j=0; j < BListSize (BucketI); j++)
        {
            Var = (GNL_VAR) BListElt (BucketI, j);
            if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
                         (GnlVarDir (Var) != GNL_VAR_INOUT)) )
                continue;
        }
    }
}

```

```

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))
        return (GnlStatus);

    IsEndInput = IsEndSeq = 0;
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);

    if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -
        GnlTimingInfoArrivalFall (TimingInfo))
        Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                GnlTimingInfoArrivalFall (TimingInfo);
    if (!GnlFloatEpsilonEqual (MinSeqSlack, Slack, MinSeqSlack * AuxEpsi))
        continue;

    if (GnlGetEpsiCritInstFromVar (Var, Lib, ListCritInst, Epsi,
                                   &IsEndInput, &IsEndSeq, MinSeqSlack, 0, 1, 0))
        return (GNL_MEMORY_FULL);
    }

}

if (GnlGetSeqEpsiCriticInstancesFromFFInGnl (Gnl, Lib, MinSeqSlack,
                                              ListCritInst, Epsi))
    return (GnlStatus);

GnlPrintCriticalInst (ListCritInst, AreaCritRegion);
for (i=0; i < BListSize (ListCritInst); i++)
{
    if (!BListElt (ListCritInst, i))
        continue;
    BListDelete ((BLIST*) &BListElt (ListCritInst, i), GnlFreeCritPath);
}
BListQuickDelete (&ListCritInst);
return (GNL_OK);
}

/*-----*/
/* GnlGetEpsiCriticInstancesFromGnl */
/*-----*/
/* Doc procedure GnlGetEpsiCriticInstancesFromGnl :
   - For a given Gnl (GNL) :
     - Get the critical region (set of epsilon-critical instances.
*/
/*-----*/
GNL_STATUS GnlGetEpsiCriticInstancesFromGnl (GNL Gnl, LIBC_LIB Lib,
                                              float MinSeqSlack,
                                              float MinCombSlack,
                                              double AreaWithoutNet)
{
    GNL_STATUS      GnlStatus;
    float          Epsi;
    double         AreaCritRegion;

    Epsi = 0.2;
    AreaCritRegion = 0.0;
}

```

timeutil.c

```
if (GnlGetCombEpsiCriticInstancesFromGnl (Gnl, Lib, MinCombSlack,
                                         Epsi, &AreaCritRegion))
    return (GNL_MEMORY_FULL);

if (GnlGetSeqEpsiCriticInstancesFromGnl (Gnl, Lib, MinSeqSlack,
                                         Epsi, &AreaCritRegion))
    return (GNL_MEMORY_FULL);

fprintf (stderr,
        " ----- \n");
fprintf (stderr, " Critical Gates Area / Total Gates Area = %.2f %s\n",
        AreaCritRegion / AreaWithoutNet * 100, "%");
fprintf (stderr,
        " ----- \n");
return (GNL_OK);
}
/*-----*/
/*-----*/
```

```
/*
/* GnlCreateClock */
/*
GNL_STATUS GnlCreateClock (GNL_CLOCK *Clock)
{
    if ((*Clock = (GNL_CLOCK)
         calloc (1, sizeof(GNL_CLOCK_REC))) == NULL)
        return (GNL_MEMORY_FULL);

    SetGnlClockSourceClock (*Clock, NULL);
    SetGnlClockHierNameOfGnlClock (*Clock, NULL);
    SetGnlClockArrTime (*Clock, 0.0);
    SetGnlClockReqTime (*Clock, 0.0);
    SetGnlClockSeqInst (*Clock, NULL);
    SetGnlClockSeqHierInstNames (*Clock, NULL);

    return (GNL_OK);
}

/*
/* GnlAddSeqInstToClock */
/*
GNL_STATUS GnlAddSeqInstToClock (GNL_CLOCK Clock,
                                  GNL_SEQUENTIAL_COMPONENT SeqCompo,
                                  char *HierInstName)
{
    BLIST      ListSeqInstances, ListSeqHierInstNames;
    BLIST      SubList1, SubList2;
    unsigned int Key;

    if (!Clock || !SeqCompo)
        return (GNL_OK);

    if (!GnlClockSeqInst (Clock))
    {
        if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &ListSeqInstances))
```

timeutil.c

```
    return (GNL_MEMORY_FULL);
BSize (ListSeqInstances) = HASH_LIST_PATH_ELEM;
SetGnlClockSeqInst (Clock, ListSeqInstances);

if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &ListSeqHierInstNames))
    return (GNL_MEMORY_FULL);
BSize (ListSeqHierInstNames) = HASH_LIST_PATH_ELEM;
SetGnlClockSeqHierInstNames (Clock, ListSeqHierInstNames);
}

ListSeqInstances = GnlClockSeqInst (Clock);
ListSeqHierInstNames = GnlClockSeqHierInstNames (Clock);

Key = KeyOfName (HieraInstName, HASH_LIST_PATH_ELEM);
SubList1 = (BLIST) BListElt (ListSeqHierInstNames, Key);
SubList2 = (BLIST) BListElt (ListSeqInstances, Key);
if (!SubList1)
{
    if (BListCreateWithSize (1, &SubList1))
        return (GNL_MEMORY_FULL);
    BListElt (ListSeqHierInstNames, Key) = (int) SubList1;
    if (BlistCreateWithSize (1, &SubList2))
        return (GNL_MEMORY_FULL);
    BListElt (ListSeqInstances, Key) = (int) SubList2;
}

if (BListAddElt (SubList2, SeqCompo))
    return (GNL_MEMORY_FULL);
if (BListAddElt (SubList1, HieraInstName))
    return (GNL_MEMORY_FULL);

return (GNL_OK);
}

/*-----
/* StrFree
/*-----
void StrFree (char **Str)
{
    if (!(*Str))
        return;

    free (*Str);
}
/*-----
/* GnlFreeClock
/*-----
void GnlFreeClock (GNL_CLOCK *Clock)
{
    int             i, Rank;
    BLIST           SubList1, SubList2;

    if (!(*Clock))
        return;

    for (i=0; i < BListSize (GnlClockSeqHierInstNames (*Clock)); i++)
    {
```

timeutil.c

```
SubList1 = (BLIST) BListElt (GnlClockSeqHierInstNames (*Clock), i);
SubList2 = (BLIST) BListElt (GnlClockSeqInst (*Clock), i);
if (!SubList1)
    continue;
BListDelete (&SubList1, StrFree);
BListQuickDelete (&SubList2);
}
BListQuickDelete (&GnlClockSeqInst (*Clock));
BListQuickDelete (&GnlClockSeqHierInstNames (*Clock));

if (GnlClockHierNameOfGnlClock (*Clock))
    free (GnlClockHierNameOfGnlClock (*Clock));

free ((char *) *Clock);
(*Clock) = NULL;
}
/*-----*/
int GnlClock_ClockCmpClock (GNL_CLOCK Clock, GNL_VAR SourceClock)
{
    if (!Clock || !SourceClock)
        return (0);

    if (GnlClockSourceClock (Clock) != SourceClock)
        return (0);

    return (1);
}

/*-----*/
/* GnlGetSourceClockFromUserCompo */
/*-----*/
GNL_STATUS GnlGetSourceClockFromUserCompo (GNL_ASSOC Assoc,
                                             GNL_VAR *SourceClock,
                                             BLIST *PileOfInstCompo,
                                             GNL_VAR Actual)
{
    GNL_VAR          Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_STATUS       GnlStatus;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);

    Formal = GnlAssocFormalPort (Assoc);
    if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
    {
        /* It corresponds to a LIBC library cell. */
        if (GnlUserComponentCellDef (UserCompo))
        {
            *SourceClock = Var;
            if (BListCopyNoEltCr (G_PileOfComponent, PileOfInstCompo))
                return (GNL_MEMORY_FULL);

            return (GNL_OK);
        }
    }
}
```

timeutil.c

```
}

else
/* It is a black box without any definition so we stop */
/* at its boundary.
{
    fprintf (stderr,
        "# WARNING: black box <%s %s> may modify final estimation\n",
        GnlUserComponentName (UserCompo),
        GnlUserComponentInstName (UserCompo));
}

return (GNL_OK);
}

if (GnlVarDir (Formal) == GNL_VAR_OUTPUT)
{
    if (BListAddElt (G_PileOfComponent, UserCompo))
        return (GNL_MEMORY_FULL);

    if (GnlStatus = TimeGetFormalFromAssocAndActual (Assoc, Actual,
&AuxFormal))
        return (GnlStatus);

    if (GnlStatus = GnlGetSourceVarFromVar (AuxFormal, SourceClock,
                                                PileOfInstCompo))
        return (GnlStatus);
    BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
}
else
{
    if (GnlStatus = GnlGetSourceVarFromVar (Actual, SourceClock,
                                                PileOfInstCompo))
        return (GnlStatus);
}

return (GNL_OK);
}

/*-----*/
/* GnlGetSourceClockFromAssoc */
/*-----*/
GNL_STATUS GnlGetSourceClockFromAssoc (GNL_ASSOC Assoc, GNL_VAR *SourceClock,
                                         BLIST *PileOfInstCompo, GNL_VAR Actual)
{
    GNL_STATUS          GnlStatus;
    int                 Rank;
    GNL_COMPONENT       GnlCompo;
    GNL_VAR             Clock;

    GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

    switch (GnlComponentType (GnlCompo)) {

        case GNL_SEQUENTIAL_COMPO:
            Clock = GnlAssocActualPort (Assoc);
            *SourceClock = Clock;
            if (BListCopyNoEltCr (G_PileOfComponent, PileOfInstCompo))
                return (GNL_MEMORY_FULL);
    }
}
```

```

        break;

    case GNL_USER_COMPO:
        if (GnlStatus = GnlGetSourceClockFromUserCompo (Assoc,
SourceClock,
                                                PileOfInstCompo, Actual))
            return (GnlStatus);
        break;

    case GNL_TRISTATE_COMPO:
        Clock = GnlAssocActualPort (Assoc);
        *SourceClock = Clock;
        if (BlistCopyNoEltCr (G_PileOfComponent, PileOfInstCompo))
            return (GNL_MEMORY_FULL);
        break;

    case GNL_MACRO_COMPO:
        break;

    default:
        GnlError (12 /* Unknown component */);
        break;
    }
    return (GNL_OK);
}
/*-----*/
/* GnlGetSourceVarFromVar */
/*-----*/
GNL_STATUS GnlGetSourceVarFromVar (GNL_VAR Clock, GNL_VAR *SourceClock,
                                    BLIST *PileOfInstCompo)
{
    GNL_VAR          VarI, Actual;
    GNL_ASSOC        AuxAssocVar;
    int              i;
    GNL_STATUS       GnlStatus;
    GNL_USER_COMPONENT InstOfGnl;
    BLIST            AssocSources, RightVarAssigneds;

    if (!Clock)
        return (GNL_OK);

    if (GnlVarDir (Clock) == GNL_VAR_INPUT || GnlVarDir (Clock) == GNL_VAR_INOUT
)
    {
        if (!BlistSize (G_PileOfComponent))
        {
            *SourceClock = Clock;
            if (BlistCreate (PileOfInstCompo))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
        InstOfGnl = (GNL_USER_COMPONENT) BlistElt (G_PileOfComponent,
                                                BlistSize (G_PileOfComponent) - 1);

        BlistDelShift (G_PileOfComponent, BlistSize (G_PileOfComponent));
    }
}

```

timeutil.c

```

AuxAssocVar
=GnlGetAssocFromFormalPortAndHierBlock(Clock, InstOfGnl, &Actual);
if (!AuxAssocVar)
{
    if (BListAddElt (G_PileOfComponent, InstOfGnl))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
if (GnlVarIsVar (Actual))
if(GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))
{
    *SourceClock = Actual;
    if (BListCopyNoEltCr (G_PileOfComponent, PileOfInstCompo))
        return (GNL_MEMORY_FULL);
    return (GNL_OK);
}
if (GnlGetSourceClockFromAssoc (AuxAssocVar, SourceClock,
                                PileOfInstCompo, Actual))
    return (GNL_MEMORY_FULL);

if (BListAddElt (G_PileOfComponent, InstOfGnl))
    return (GNL_MEMORY_FULL);
return (GNL_OK);
}

if (GnlGetSourcesOfVar (Clock, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    VarI = (GNL_VAR) BListElt (RightVarAssigneds, i);
    if (GnlVarIsVar (VarI))
    if(GnlVarIsVdd (VarI) || GnlVarIsVss (VarI))
    {
        *SourceClock = VarI;
        if (BListCopyNoEltCr (G_PileOfComponent, PileOfInstCompo))
            return (GNL_MEMORY_FULL);
        BListQuickDelete (&RightVarAssigneds);
        BListQuickDelete (&AssocSources);
        return (GNL_OK);
    }
    if (GnlGetSourceVarFromVar (VarI, SourceClock, PileOfInstCompo))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&RightVarAssigneds);
for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    if (GnlGetSourceClockFromAssoc (AuxAssocVar, SourceClock,
                                    PileOfInstCompo, Clock))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&AssocSources);
return (GNL_OK);
}

```

```

/*-----*/
/*-----*/
/* GnlGetGetClocksFromGnl */
/*-----*/
/* Doc procedure GnlGetGetClocksFromGnl :
   - Getting all clock (GNL_CLOCK) from a given netlist (GNL).

/*-----*/
GNL_STATUS GnlGetGetClocksFromGnl (GNL Gnl, BLIST ListOfClocks)
{
    GNL_VAR             Var, SourceClock;
    int                 i, Rank;
    GNL_STATUS          GnlStatus;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT       ComponentI;
    GNL                 GnlCompoI;
    GNL_ASSOC           Assoc;
    char                *HierNameOfGnlClock, *HieraInstName;
    GNL_CLOCK           Clock;
    BLIST               PileOfInstCompo;
    BLIST               SaveG_PileOfComponent;

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
            {
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                    return (GNL_MEMORY_FULL);
                if (GnlStatus = GnlGetGetClocksFromGnl (GnlCompoI, ListOfClocks))
                    return (GnlStatus);
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            }
        }
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
            continue;

        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
        Assoc = GnlSequentialCompoClockAssoc (SeqCompo);
        if (!Assoc)
            continue;
        Var = GnlAssocActualPort (Assoc);
        if (!Var)
            continue;
        if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;
        if (GnlStatus = GnlGetSourceVarFromVar (Var, &SourceClock,
                                                &PileOfInstCompo))
            return (GnlStatus);

        if (Rank = BListMemberOfList (ListOfClocks, SourceClock,
                                     GnlClock_ClockCmpClock))

```

```

{
    Clock = (GNL_CLOCK) BListElt (ListOfClocks, Rank-1);
}
else
{
    if (GnlCreateClock (&Clock))
        return (GNL_MEMORY_FULL);
    SetGnlClockSourceClock (Clock, SourceClock);
    SaveG_PileOfComponent = G_PileOfComponent;
    G_PileOfComponent = PileOfInstCompo;
    if (GnlGetHierarchycalInstName ((char *) NULL, &HierNameOfGnlClock))
        return (GNL_MEMORY_FULL);
    G_PileOfComponent = SaveG_PileOfComponent;
    SetGnlClockHierNameOfGnlClock (Clock, HierNameOfGnlClock);
    if (BListAddElt (ListOfClocks, Clock))
        return (GNL_MEMORY_FULL);
}
BListQuickDelete (&PileOfInstCompo);
GnlGetHierarchycalInstName (GnlSequentialCompoInstName (SeqCompo),
                            &HieraInstName);
GnlAddSeqInstToClock (Clock, SeqCompo, HieraInstName);
}

return (GNL_OK);
}

/*-----*/
/* GnlGetClocksFromNetwork */
/*-----*/
/* Doc procedure GnlGetClocksFromNetwork :
   - Getting all clock (GNL_CLOCK) from a given
     Network (GNL_NETWORK).
*/
/*-----*/
GNL_STATUS GnlGetClocksFromNetwork (GNL_NETWORK Nw, BLIST ListOfClocks,
                                     int PrintClkDomain)
{
    GNL          TopGnl;
    GNL_STATUS   GnlStatus;
    GNL_CLOCK    Clock;
    GNL_VAR      SourceClock;
    BLIST       ListSeqhierInstNames, ListSeqInsts;
    char         *Name;
    GNL_SEQUENTIAL_COMPONENT Seq;
    int          i, j, k;
    BLIST       SubList1, SubList2;

    TopGnl = GnlNetworkTopGnl (Nw);

    if (BListCreate (&G_PileOfComponent))
        return (GNL_MEMORY_FULL);

    if ((GnlStatus = GnlGetGetClocksFromGnl (TopGnl, ListOfClocks)))
        return (GnlStatus);

    BListQuickDelete (&G_PileOfComponent);
}

```

```

if (PrintClkDomain)
for (i=0; i < BListSize (ListOfClocks); i++)
{
    Clock = (GNL_CLOCK) BListElt (ListOfClocks, i);
    SourceClock = GnlClockSourceClock (Clock);
    fprintf (stderr,
             "\n -----\n");
    fprintf (stderr, " Sequential Components Driven by \"%s\" :\n\n",
             GnlVarName (SourceClock));
    ListSeqHierInstNames = GnlClockSeqHierInstNames (Clock);
    ListSeqInsts = GnlClockSeqInst (Clock);
    for (j=0; j < BListSize (ListSeqHierInstNames); j++)
    {
        SubList1 = (BLIST) BListElt (ListSeqHierInstNames, j);
        SubList2 = (BLIST) BListElt (ListSeqInsts, j);
        if (!SubList1)
            continue;
        for (k=0; k < BListSize (SubList1); k++)
        {
            Name = (char *) BListElt (SubList1, k);
            Seq = (GNL_SEQUENTIAL_COMPONENT) BListElt (SubList2, k);
            fprintf (stderr, "\t\t%s", Name);
            fprintf (stderr, " (%s)\n", LibCellName (GnlSeqCompoInfoCell
(Seq)));
        }
    }
    fprintf (stderr,
             "\n -----\n");
}

return (GNL_OK);
}

/* ----- EOF ----- */

/*
/* TimeOptByResizingCombCell
*/
GNL_STATUS TimeOptByResizingCombCell (GNL_USER_COMPONENT UserCompo,
                                       GNL_ASSOC      Assoc,
                                       LIBC_LIB       Lib,
                                       GNL           TopGnl)
{
    int          i, j;
    LIB_DERIVE_CELL DriveCell, CurrentDeriveCell, DeriveCellI;
    LIB_CELL      MotherCell, CurrentMotherCell, MotherCellI;
    LIBC_CELL     Cell;

    float         MaxArrivalTime, MaxJ, Max;
    float         Length, WireResistance, WireCapa, OutCapa;
    float         InTransRise, InTransFall, OutTransR, OutTransF;
    int          Fanout, Rank, ReplaceWasNotOk;
    GNL_TIMING_INFO TimingJ, Timing;
    char          *Formal;
    LIBC_PIN      PinOut, PinIn;
    GNL_ASSOC     AssocJ;
}

```

timeutil.c

```
GNL_VAR Var, VarJ;
BLIST ListEquivCells, Interface;
unsigned int Key;
float DelayR, DelayF, TimeRise, TimeFall;
Gnl Gnl;
GNL_USER_COMPONENT InstOfGnl;

Gnl = TopGnl;
if (BListSize (G_PileOfComponent))
{
    InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                BlistSize (G_PileOfComponent)-1);
    Gnl = GnlUserComponentGnlDef (InstOfGnl);
}
ListEquivCells = GnlUserComponentEquivCells (UserCompo);
if (!ListEquivCells || !BListSize (ListEquivCells))
    return (GNL_OK);

Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
Var = GnlAssocActualPort (Assoc);
Formal = (char *) GnlAssocFormalPort (Assoc);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &Timing, &Key, &Rank))
    return (GNL_MEMORY_FULL);

MaxArrivalTime = GnlTimingInfoArrivalRise (Timing);
if (MaxArrivalTime < GnlTimingInfoArrivalFall (Timing))
    MaxArrivalTime = GnlTimingInfoArrivalFall (Timing);
Fanout = GnlTimingInfoFanout (Timing);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (Timing);

GnlFindCellsUserCompo (UserCompo, &CurrentMotherCell,
                       &CurrentDeriveCell);

for (i=0; i < BListSize (ListEquivCells); i++)
{
    DriveCell = (LIB_DERIVE_CELL) BListElt (ListEquivCells, i);
    /* We dont accept the inverter cell */
    if (LibDeriveCellBdd (CurrentDeriveCell) !=
        LibDeriveCellBdd (DriveCell))
        continue;
    MotherCell = LibDeriveCellMotherCell (DriveCell);
    Cell = LibHCellLibcCell (MotherCell);
    if (LibCellPad (Cell) || LibCellDontTouch (Cell) || LibCellDontUse
        (Cell))
        continue;
    PinOut = LibGetPinFromNameAndCell (Cell, LibHCellOutput (MotherCell));
    if (GnlReplaceUserCompoByDeriveCell (Gnl, UserCompo, DriveCell,
                                         (LIB_DERIVE_CELL) NULL,
                                         (BLIST) NULL))
        return (GNL_MEMORY_FULL);
    ReplaceWasNotOk = 0;
    Interface = GnlUserComponentInterface (UserCompo);
    Max = -MAX_FLOAT;
```

timeutil.c

```
for (j=0; j < BListSize (Interface); j++)
{
    AssocJ = (GNL_ASSOC)BListElt (Interface, j);
    VarJ = GnlAssocActualPort (AssocJ);
    Formal = (char *) GnlAssocFormalPort (AssocJ);
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinIn) != INPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarJ,&TimingJ,&Key,&Rank))
        return (GNL_MEMORY_FULL);
    InTransRise = GnlTimingInfoTransitionRise (TimingJ);
    InTransFall = GnlTimingInfoTransitionFall (TimingJ);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                    Fanout, Cell, PinIn, PinOut,
                    &DelayR, &OutTransR,
                    &DelayF, &OutTransF);
    GnlGetArrivalTimeRiseAndFallSwitchTimingsence (PinIn, PinOut,
                                                    DelayR, DelayF,
                                                    GnlTimingInfoArrivalRise (TimingJ),
                                                    GnlTimingInfoArrivalFall (TimingJ),
                                                    &TimeRise, &TimeFall);

    MaxJ = TimeRise;
    if (MaxJ < TimeFall)
        MaxJ = TimeFall;
    if (Max < MaxJ)
        Max = MaxJ;
    if (Max > MaxArrivalTime)
    {
        ReplaceWasNotOk = 1;
        break;
    }
}

if (ReplaceWasNotOk)
{
    if (GnlReplaceUserCompoByDeriveCell (Gnl, UserCompo,
                                         CurrentDeriveCell,
                                         (LIB_DERIVE_CELL) NULL,
                                         (BLIST) NULL))
        return (GNL_MEMORY_FULL);
}
else
{
    CurrentDeriveCell = DriveCell;
    MaxArrivalTime = Max;
}
}

return (GNL_OK);
}

/*-----*/
```

timeutil.c

```
/* TimeOptByResizingFromCombCell */  
/*-----*/  
GNL_STATUS TimeOptByResizingFromCombCell (GNL_USER_COMPONENT UserCompo,  
                                         GNL_ASSOC Assoc,  
                                         LIBC_LIB Lib,  
                                         float Epsi,  
                                         float MinSlack,  
                                         BLIST ListCritInst,  
                                         GNL TopGnl,  
                                         BLIST ListBuffers,  
                                         LIBC_CELL Buffer,  
                                         int *Tag,  
                                         int *Id,  
                                         int IsAboutFanout)  
{  
    LIBC_CELL Cell, AuxCell;  
    LIBC_PIN PinOut, PinIn;  
    int i, Rank, Fanout, IsEndInputAux, IsEndSeqAux;  
    GNL_ASSOC AssocI;  
    char *Formal, *HierarchycalName;  
    GNL_VAR Var, VarI;  
    GNL_TIMING_INFO TimingI, Timing;  
    float InTransRise, InTransFall, OutTransR, OutTransF,  
          DelayR, DelayF, TimeRise, TimeFall;  
    float Length, WireCapa, WireResistance, OutCapa;  
    float Max, MaxI;  
    BLIST Interface, SubList;  
    unsigned int Key;  
  
    Cell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);  
    Var = GnlAssocActualPort (Assoc);  
    Formal = (char *) GnlAssocFormalPort (Assoc);  
  
    if (!(PinOut = LibGetPinFromNameAndCell (Cell, Formal)))  
        return (GNL_OK);  
    if (LibPinDirection (PinOut) == INPUT_E )  
        return (GNL_OK);  
  
    if (GnlGetHierarchycalInstName (GnlUserComponentInstName (UserCompo),  
                                    &HierarchycalName))  
        return (GNL_MEMORY_FULL);  
    Key = KeyOfName (HierarchycalName, BListSize (ListCritInst));  
    SubList = (BLIST)BListElt (ListCritInst, Key);  
    if (!SubList)  
    {  
        if (BListCreate (&SubList))  
            return (GNL_MEMORY_FULL);  
        BListElt (ListCritInst, Key) = (int) SubList;  
    }  
    if (BListMemberOfList (SubList, HierarchycalName, StringIdentical))  
    {  
        free (HierarchycalName);  
        return (GNL_OK);  
    }  
    if (BListAddElt (SubList, (int) HierarchycalName))  
        return (GNL_MEMORY_FULL);
```

```

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &Timing, &Key, &Rank))
    return (GNL_MEMORY_FULL);

Max = GnlTimingInfoArrivalRise (Timing);
if (Max < GnlTimingInfoArrivalFall (Timing))
    Max = GnlTimingInfoArrivalFall (Timing);
Fanout = GnlTimingInfoFanout (Timing);
Length = LibGetWireLengthFromFanout (G_WireLoad, Fanout);
WireResistance = LibGetWireScaledResiFromLength (G_WireLoad, Length, Lib);
WireCapa = LibGetWireScaledCapaFromLength (G_WireLoad, Length, Lib);
OutCapa = WireCapa + GnlTimingInfoCapa (Timing);

Interface = GnlUserComponentInterface (UserCompo);

for (i=0; i < BListSize (Interface); i++)
{
    AssocI = (GNL_ASSOC)BListElt (Interface, i);
    VarI = GnlAssocActualPort (AssocI);
    if (GnlVarIsVss (VarI) || GnlVarIsVdd (VarI))
        continue;
    Formal = (char *) GnlAssocFormalPort (AssocI);
    if (!(PinIn = LibGetPinFromNameAndCell (Cell, Formal)))
        continue;
    if (LibPinDirection(PinIn) != INPUT_E)
        continue;
    if (!LibGetArcTiming (PinIn, PinOut))
        continue;
    if (GnlTimingInfoCorrespToCurrentPathFromVar
(VarI,&TimingI,&Key,&Rank))
        return (GNL_MEMORY_FULL);
    InTransRise = GnlTimingInfoTransitionRise (TimingI);
    InTransFall = GnlTimingInfoTransitionFall (TimingI);

    LibDelayArcCell (InTransRise, InTransFall, OutCapa, WireResistance,
                    Fanout, Cell, PinIn, PinOut,
                    &DelayR, &OutTransR,
                    &DelayF, &OutTransF);
    GnlGetArrivalTimeRiseAndFallSwitchTimingSence (PinIn, PinOut,
                                                   DelayR, DelayF,
                                                   GnlTimingInfoArrivalRise (TimingI),
                                                   GnlTimingInfoArrivalFall (TimingI),
                                                   &TimeRise, &TimeFall);
    MaxI = TimeRise;
    if (MaxI < TimeFall)
        MaxI = TimeFall;
    if (!GnlFloatEpsilonEqual (Max, MaxI, Max * Epsi))
        continue;
    if (TimeOptByResizingFromVar (VarI, Lib, Epsi, MinSlack, 1,
                                  ListCritInst, TopGnl, ListBuffers,
                                  Buffer, Tag, Id, IsAboutFanout))
        return (GNL_MEMORY_FULL);
}

if (!IsAboutFanout)
{
    if (TimeOptByResizingCombCell (UserCompo, Assoc, Lib, TopGnl))

```

timeutil.c

```
    return (GNL_MEMORY_FULL);

/* AuxCell = (LIBC_CELL) GnlUserComponentCellDef (UserCompo);
if (Cell != AuxCell)
{
    fprintf (stderr, "WARNING: Replacing <%s> Cell <%s>",
GnlUserComponentInstName (UserCompo), LibCellName (Cell));
    fprintf (stderr, "by Cell <%s>\n", LibCellName (AuxCell));
} */

if (GnlGetHierarchycalInstName (GnlUserComponentInstName (UserCompo),
                                &HierarchycalName))
    return (GNL_MEMORY_FULL);

Key = KeyOfName (HierarchycalName, BListSize (ListCritInst));
SubList = (BLIST)BListElt (ListCritInst, Key);
if (!SubList)
{
    if (BListCreate (&SubList))
        return (GNL_MEMORY_FULL);
    BListElt (ListCritInst, Key) = (int) SubList;
}
if (!BListMemberOfList (SubList, HierarchycalName, StringIdentical))
    if (BListAddElt (SubList, (int) HierarchycalName))
        return (GNL_MEMORY_FULL);
}
return (GNL_OK);
}

/*-----*/
/* TimeOptByResizingFromUserCompo */
/*-----*/
GNL_STATUS TimeOptByResizingFromUserCompo (GNL_ASSOC Assoc,
                                             LIBC_LIB     Lib,
                                             float        Epsi,
                                             float        MinSlack,
                                             GNL_VAR      Actual,
                                             BLIST        ListCritInst,
                                             GNL          TopGnl,
                                             BLIST        ListBuffers,
                                             LIBC_CELL    Buffer,
                                             int          *Tag,
                                             int          *Id,
                                             int          IsAboutFanout)
{
    GNL_VAR      Var, Formal, AuxFormal;
    GNL_USER_COMPONENT UserCompo;
    GNL_TIMING_INFO TimingInfo;
    GNL_STATUS    GnlStatus;
    int          Rank;

    UserCompo = (GNL_USER_COMPONENT) GnlAssocTraversalInfoComponent (Assoc);
    Var = GnlAssocActualPort (Assoc);

    Formal = GnlAssocFormalPort (Assoc);
```

timeutil.c

```
if (GnlUserComponentFormalType (UserCompo) == GNL_FORMAL_CHAR)
{
    /* It corresponds to a LIBC library cell. */
    if (GnlUserComponentCellDef (UserCompo))
    {
        if (TimeOptByResizingFromCombCell (UserCompo, Assoc, Lib, Epsi,
MinSlack,
                                         ListCritInst, TopGnl, ListBuffers,
                                         Buffer, Tag, Id, IsAboutFanout))
            return (GNL_MEMORY_FULL);

        return (GNL_OK);
    }
    else
    /* It is a black box without any definition so we stop */
    /* at its boundary. */
    {
        fprintf (stderr,
        "# WARNING: black box <%s %s> may modify final estimation\n",
        GnlUserComponentName (UserCompo),
        GnlUserComponentInstName (UserCompo));
    }
    return (GNL_OK);
}

if (BListAddElt (G_PileOfComponent, UserCompo))
    return (GNL_MEMORY_FULL);

if (TimeGetFormalFromAssocAndActual (Assoc, Actual, &AuxFormal))
    return (GNL_MEMORY_FULL);

if (TimeOptByResizingFromVar (AuxFormal, Lib, Epsi, MinSlack, 0,
                               ListCritInst, TopGnl, ListBuffers,
                               Buffer, Tag, Id, IsAboutFanout))
    return (GNL_MEMORY_FULL);

BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
return (GNL_OK);
}
/*-----*/
/* TimeOptByResizingFromAssoc */
/*-----*/
GNL_STATUS TimeOptByResizingFromAssoc (GNL_ASSOC          Assoc,
                                         LIBC_LIB      Lib,
                                         float         Epsi,
                                         float         MinSlack,
                                         GNL_VAR       Actual,
                                         BLIST         ListCritInst,
                                         GNL           TopGnl,
                                         BLIST         ListBuffers,
                                         LIBC_CELL     Buffer,
                                         int           *Tag,
                                         int           *Id,
                                         int           IsAboutFanout)
{
    GNL_STATUS      GnlStatus;
    int             Rank;
```

timeutil.c

```
GNL_COMPONENT           GnlCompo;

GnlCompo = GnlAssocTraversalInfoComponent (Assoc);

switch (GnlComponentType (GnlCompo)) {

    case GNL_SEQUENTIAL_COMPO:
        break;

    case GNL_USER_COMPO:
        if (TimeOptByResizingFromUserCompo (Assoc, Lib, Epsi,
                                             MinSlack, Actual,
                                             ListCritInst, TopGnl,
                                             ListBuffers, Buffer, Tag,
                                             Id, IsAboutFanout))
            return (GNL_MEMORY_FULL);
        break;

    case GNL_TRISTATE_COMPO:
        break;

    case GNL_MACRO_COMPO:
        break;

    default:
        break;
}
return (GNL_OK);
}

/*-----*/
/* TimeOptByResizingFromVar */ */
/*-----*/
GNL_STATUS TimeOptByResizingFromVar (GNL_VAR      Var,
                                      LIBC_LIB     Lib,
                                      float        Epsi,
                                      float        MinSlack,
                                      int          InoutIsIn,
                                      BLIST        ListCritInst,
                                      GNL          TopGnl,
                                      BLIST        ListBuffers,
                                      LIBC_CELL    Buffer,
                                      int          *Tag,
                                      int          *Id,
                                      int          IsAboutFanout)
{
    GNL_VAR      VarI, Actual;
    GNL_ASSOC    AuxAssocVar;
    int          i, Rank;
    GNL_STATUS   GnlStatus;
    GNL_TIMING_INFO Timing, TimingI;
    float         ArrivalTime, RequiredTime,
                  SlackRise, SlackFall, Slack, Max, MaxI;
    GNL_USER_COMPONENT InstOfGnl;
    BLIST        AssocSources, RightVarAssigneds;
    unsigned int  Key;
```

```

if (GnlVarIsVar (Var))
    if (GnlVarIsVdd (Var) || GnlVarIsVss (Var))
        return (GNL_OK);

if (IsAboutFanout)
if (TimeOptBalanceRequiredTime (Var, Buffer, Lib, Id, TopGnl, Tag,
                               InoutIsIn, ListBuffers))
    return (GNL_MEMORY_FULL);

if (GnlTimingInfoCorrespToCurrentPathFromVar (Var,&Timing, &Key, &Rank))
    return (GNL_MEMORY_FULL);

if (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn)
    Timing = (GNL_TIMING_INFO) GnlTimingInfoHook (Timing);

    Max = GnlTimingInfoArrivalRise (Timing);
    if (Max < GnlTimingInfoArrivalFall (Timing))
        Max = GnlTimingInfoArrivalFall (Timing);

if (GnlVarDir (Var) == GNL_VAR_INPUT ||
    (GnlVarDir (Var) == GNL_VAR_INOUT && InoutIsIn))
{
    if (!BListSize (G_PileOfComponent))
    {
        return (GNL_OK);
    }
    else
    {
        InstOfGnl = (GNL_USER_COMPONENT) BListElt (G_PileOfComponent,
                                                    BListSize (G_PileOfComponent)-1);

        BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
        AuxAssocVar = GnlGetAssocFromFormalPortAndHierBlock (Var, InstOfGnl,
                                                              &Actual);
        if (!AuxAssocVar)
        {
            if (BListAddElt (G_PileOfComponent, InstOfGnl))
                return (GNL_MEMORY_FULL);
            return (GNL_OK);
        }
        if (!(GnlVarIsVar (Actual) &&
              (GnlVarIsVdd (Actual) || GnlVarIsVss (Actual))))
        {
            if (TimeOptByResizingFromVar (Actual, Lib, Epsi, MinSlack, 1,
                                           ListCritInst, TopGnl, ListBuffers,
                                           Buffer, Tag, Id, IsAboutFanout))
                return (GNL_MEMORY_FULL);
        }
        if (BListAddElt (G_PileOfComponent, InstOfGnl))
            return (GNL_MEMORY_FULL);
    }
}

if (GnlGetSourcesOfVar (Var, &AssocSources, &RightVarAssigneds))
    return (GNL_MEMORY_FULL);

```

timeutil.c

```
for (i=0; i < BListSize (RightVarAssigneds); i++)
{
    VarI = (GNL_VAR) BListElt (RightVarAssigneds, i);

    if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (VarI,
                                                               &TimingI, &Key, &Rank)))
        return (GnlStatus);
    MaxI = GnlTimingInfoArrivalRise (TimingI);
    if (MaxI < GnlTimingInfoArrivalFall (TimingI))
        MaxI = GnlTimingInfoArrivalFall (TimingI);
    if (GnlFloatEpsilonEqual (Max, MaxI, Max * Epsi) )
        if (TimeOptByResizingFromVar (VarI, Lib, Epsi, MinSlack, 1,
                                       ListCritInst, TopGnl, ListBuffers,
                                       Buffer, Tag, Id, IsAboutFanout))
            return (GNL_MEMORY_FULL);
    }

BListQuickDelete (&RightVarAssigneds);

for (i=0; i < BListSize (AssocSources); i++)
{
    AuxAssocVar = (GNL_ASSOC) BListElt (AssocSources, i);
    if (TimeOptByResizingFromAssoc (AuxAssocVar, Lib, Epsi, MinSlack, Var,
                                    ListCritInst, TopGnl, ListBuffers,
                                    Buffer, Tag, Id, IsAboutFanout))
        return (GNL_MEMORY_FULL);
}

BListQuickDelete (&AssocSources);

return (GNL_OK);
}

/*-----*/
/* TimeOptByResizingFromMaxCombSlackAndGnl */
/*-----*/
GNL_STATUS TimeOptByResizingFromMaxCombSlackAndGnl (
    GNL          Gnl,
    LIBC_LIB     Lib,
    float        MinCombSlack,
    float        Epsi,
    BLIST        ListBuffers,
    LIBC_CELL    Buffer,
    int          *Tag,
    int          *Id,
    int          IsAboutFanout)
{
    GNL_VAR      Var;
    int          i, j, Rank;
    GNL_STATUS   GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float         MinSlack, Slack;
    BLIST         BucketI;
    unsigned int  Key;
    float         AuxEpsi;
    BLIST         ListCritInst;
```

```

if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &ListCritInst))
    return (GNL_MEMORY_FULL);
BSize (ListCritInst) = HASH_LIST_PATH_ELEM;

AuxEpsi = Epsi;
if (MinCombSlack < 0.0)
    AuxEpsi = Epsi * (-1);

for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
{
    BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
    for (j=0; j < BListSize (BucketI); j++)
    {
        Var = (GNL_VAR) BListElt (BucketI, j);
        if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&
                     (GnlVarDir (Var) != GNL_VAR_INOUT)) )
            continue;

        if (GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                       &TimingInfo, &Key, &Rank))
            return (GNL_MEMORY_FULL);

        Slack = GnlTimingInfoRequiredRise (TimingInfo) -
                GnlTimingInfoArrivalRise (TimingInfo);

        if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -
            GnlTimingInfoArrivalFall (TimingInfo))
            Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                    GnlTimingInfoArrivalFall (TimingInfo);
        if (!GnlFloatEpsilonEqual (MinCombSlack, Slack, MinCombSlack * AuxEpsi))
            continue;

        if (TimeOptByResizingFromVar (Var, Lib, Epsi, MinCombSlack, 0,
                                       ListCritInst, Gnl, ListBuffers,
                                       Buffer, Tag, Id, IsAboutFanout))
            return (GNL_MEMORY_FULL);
    }
}

for (i=0; i < BListSize (ListCritInst); i++)
{
    if (!BListElt (ListCritInst, i))
        continue;
    BListDelete ((BLIST*) &BListElt (ListCritInst, i), StrFree);
}
BListQuickDelete (&ListCritInst);

return (GNL_OK);
}
/*-----*/
/* TimeOptByResizingFromFFInGnl */
/*-----*/
GNL_STATUS TimeOptByResizingFromFFInGnl (GNL           Gnl,
                                         LIBC_LIB     Lib,
                                         float        MinSeqSlack,
                                         
```

```

        float      Epsi,
        BLIST     ListCritInst,
        GNL       TopGnl,
        BLIST     ListBuffers,
        LIBC_CELL Buffer,
        int       *Tag,
        int       *Id,
        int       IsAboutFanout)
{
    GNL_VAR          Var;
    int             i, Rank, IsEndInput, IsEndSeq;
    GNL_STATUS       GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float           Slack;
    GNL_SEQUENTIAL_COMPONENT SeqCompo;
    GNL_COMPONENT    ComponentI;
    GNL             GnlCompoI;
    GNL_ASSOC        Assoc;
    unsigned int     Key;
    float           AuxEpsi;

    AuxEpsi = Epsi;
    if (MinSeqSlack < 0.0)
        AuxEpsi = Epsi * (-1);

    for (i=0; i < BListSize (GnlComponents (Gnl)); i++)
    {
        ComponentI = (GNL_COMPONENT) BListElt (GnlComponents (Gnl), i);
        if (GnlComponentType (ComponentI) == GNL_USER_COMPO)
        {
            GnlCompoI = GnlUserComponentGnlDef ((GNL_USER_COMPONENT) ComponentI);
            if (GnlCompoI)
            {
                if (BListAddElt (G_PileOfComponent, (GNL_USER_COMPONENT) ComponentI))
                    return (GNL_MEMORY_FULL);
                if (TimeOptByResizingFromFFInGnl (GnlCompoI, Lib, MinSeqSlack,
                                                   Epsi, ListCritInst, TopGnl,
                                                   ListBuffers, Buffer, Tag, Id,
                                                   IsAboutFanout))
                    return (GNL_MEMORY_FULL);
                BListDelShift (G_PileOfComponent, BListSize (G_PileOfComponent));
            }
        }
        if (GnlComponentType (ComponentI) != GNL_SEQUENTIAL_COMPO)
            continue;

        SeqCompo = (GNL_SEQUENTIAL_COMPONENT) ComponentI;
        Assoc = GnlSequentialCompoInputAssoc (SeqCompo);
        if (!Assoc)
            continue;
        Var = GnlAssocActualPort (Assoc);
        if (!Var)
            continue;
        if (GnlVarIsVss (Var) || GnlVarIsVdd (Var))
            continue;
        if ((GnlStatus = GnlTimingInfoCorrespToCurrentPathFromVar (Var,
                                                               &TimingInfo, &Key, &Rank)))

```

timeutil.c

```
    return (GnlStatus);

    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    IsEndInput = IsEndSeq = 0;
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -
            GnlTimingInfoArrivalRise (TimingInfo);
    if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -
        GnlTimingInfoArrivalFall (TimingInfo))
        Slack = GnlTimingInfoRequiredFall (TimingInfo) -
                GnlTimingInfoArrivalFall (TimingInfo);
    if (!GnlFloatEpsilonEqual (MinSeqSlack, Slack, MinSeqSlack * AuxEpsi))
        continue;

    if (TimeOptByResizingFromVar (Var, Lib, Epsi, MinSeqSlack, 1,
                                  ListCritInst, TopGnl, ListBuffers,
                                  Buffer, Tag, Id, IsAboutFanout))
        return (GNL_MEMORY_FULL);
    }

    return (GNL_OK);
}
/*-----*/
/* TimeOptByResizingFromMaxSeqSlackFromGnl */
/*-----*/
GNL_STATUS TimeOptByResizingFromMaxSeqSlackFromGnl (
    GNL          Gnl,
    LIBC_LIB     Lib,
    float        MinSeqSlack,
    float        Epsi,
    BLIST        ListBuffers,
    LIBC_CELL    Buffer,
    int          *Tag,
    int          *Id,
    int          IsAboutFanout)
{
    GNL_VAR      Var;
    int          i, j, Rank, IsEndInput, IsEndSeq;
    GNL_STATUS   GnlStatus;
    GNL_TIMING_INFO TimingInfo;
    float         MinSlack, Slack;
    BLIST         BucketI, ListCritInst;
    unsigned int  Key;
    float         AuxEpsi;

    if (BListCreateWithSize (HASH_LIST_PATH_ELEM, &ListCritInst))
        return (GNL_MEMORY_FULL);
    BSize (ListCritInst) = HASH_LIST_PATH_ELEM;

    AuxEpsi = Epsi;
    if (MinSeqSlack < 0.0)
        AuxEpsi = Epsi * (-1);

    for (i=0; i < BListSize (GnlHashNames (Gnl)); i++)
    {
        BucketI = (BLIST) BListElt (GnlHashNames (Gnl), i);
        for (j=0; j < BListSize (BucketI); j++)

```

timeutil.c

```
{  
    Var = (GNL_VAR) BListElt (BucketI, j);  
    if (!Var || ((GnlVarDir (Var) != GNL_VAR_OUTPUT) &&  
                (GnlVarDir (Var) != GNL_VAR_INOUT)) )  
        continue;  
  
    if (GnlTimingInfoCorrespToCurrentPathFromVar (Var, &TimingInfo,  
                                                &Key, &Rank))  
        return (GNL_MEMORY_FULL);  
  
    IsEndInput = IsEndSeq = 0;  
    Slack = GnlTimingInfoRequiredRise (TimingInfo) -  
            GnlTimingInfoArrivalRise (TimingInfo);  
  
    if (Slack > GnlTimingInfoRequiredFall (TimingInfo) -  
        GnlTimingInfoArrivalFall (TimingInfo))  
        Slack = GnlTimingInfoRequiredFall (TimingInfo) -  
                GnlTimingInfoArrivalFall (TimingInfo);  
    if (!GnlFloatEpsilonEqual (MinSeqSlack, Slack, MinSeqSlack * AuxEpsi))  
        continue;  
  
    if (TimeOptByResizingFromVar (Var, Lib, Epsi, MinSeqSlack, 0,  
                                ListCritInst, Gnl, ListBuffers,  
                                Buffer, Tag, Id, IsAboutFanout))  
        return (GNL_MEMORY_FULL);  
    }  
}  
if (TimeOptByResizingFromFFInGnl (Gnl, Lib, MinSeqSlack, Epsi,  
                                ListCritInst, Gnl,  
                                ListBuffers, Buffer, Tag, Id,  
                                IsAboutFanout))  
    return (GNL_MEMORY_FULL);  
  
for (i=0; i < BlistSize (ListCritInst); i++)  
{  
    if (!BListElt (ListCritInst, i))  
        continue;  
    BListDelete ((BLIST*) &BListElt (ListCritInst, i), StrFree);  
}  
BListQuickDelete (&ListCritInst);  
return (GNL_OK);  
}  
  
/*-----*/  
/* TimeOptByResizingFromGnl */  
/*-----*/  
/* Doc procedure TimeOptByResizingFromGnl :  
   - For a given Gnl (GNL) :  
     - Optimizes the critical instances by resizing the attached cells.  
*/  
/*-----*/  
GNL_STATUS TimeOptByResizingFromGnl (GNL Gnl, LIBC_LIB Lib, int  
IsAboutFanout,  
                                      int *Tag, BLIST ListBuffers,  
                                      LIBC_CELL Buffer)  
{  
    GNL_STATUS      GnlStatus;
```

timeutil.c

```
float          Epsi;
double        AreaCritRegion;
float          MinSeqSlack, MinCombSlack;
int           Id;

if (BListCreate (&G_PileOfComponent))
    return (GNL_MEMORY_FULL);

MinSeqSlack = MAX_FLOAT;
GnlGetMinSlackFromInputsOfSequentialInst (Gnl, &MinSeqSlack);
GnlGetMinSlackFromOutputs (Gnl, &MinCombSlack);

/* fprintf (stderr, "MinSeqSlack = %.3f\t MinCombSlack = %.3f\n\n",
MinSeqSlack, MinCombSlack); */

Epsi = 0.1;
Id = 0;

if (TimeOptByResizingFromMaxCombSlackAndGnl (Gnl, Lib, MinCombSlack, Epsi,
                                              ListBuffers, Buffer, Tag, &Id,
                                              IsAboutFanout))
    return (GNL_MEMORY_FULL);

if (TimeOptByResizingFromMaxSeqSlackFromGnl (Gnl, Lib, MinSeqSlack, Epsi,
                                              ListBuffers, Buffer, Tag, &Id,
                                              IsAboutFanout))
    return (GNL_MEMORY_FULL);

BListQuickDelete (&G_PileOfComponent);

return (GNL_OK);
}

/*-----*/
```

timeutil.e

```
/*-----*/
/*
 *      File:          timeutil.e
 *
 *      Version:       1.1
 *
 *      Modifications: -
 *
 *      Documentation: -
 */
/*-----*/
```

```
extern GNL_STATUS GnlCreateCritPath (GNL_CRITICAL_PATH *CritPath);

extern GNL_STATUS GnlAddPredecessorToCritPath (GNL_CRITICAL_PATH CritPath,
                                              GNL_CRITICAL_PATH Predecessor);

extern GNL_STATUS GnlAddSuccessorToCritPath (GNL_CRITICAL_PATH CritPath,
                                              GNL_CRITICAL_PATH Successor);

extern void GnlFreeCritPath (GNL_CRITICAL_PATH *CritPath);

extern GNL_STATUS GnlCreateTimingInfo (GNL_TIMING_INFO *TimingInfo);

extern void GnlFreeTimingInfo(GNL_TIMING_INFO *TimingInfo);

extern GNL_STATUS TimeCopyTimingInfo (GNL_TIMING_INFO TimingI1,
                                      GNL_TIMING_INFO *TimingI2);

extern GNL_STATUS GnlFreeTimingInfoFromNetwork (GNL_NETWORK Nw);

extern GNL_STATUS GnlGetEpsiCritInstFromVar (GNL_VAR Var,
                                             LIBC_LIB Lib,
                                             BLIST ListCritInst,
                                             float Epsi,
                                             int *IsEndInput, int *IsEndSeq,
                                             float MinSlack,
                                             int InoutIsIn,
                                             int IsStartOutput,
                                             int IsAboutComb);

extern GNL_STATUS GnlGetEpsiCriticInstancesFromGnl (GNL Gnl, LIBC_LIB Lib,
                                                    float MinSeqSlack,
                                                    float MinCombSlack,
                                                    double AreaWithoutNet);
```



```
extern GNL_STATUS GnlGetSourceVarFromVar (GNL_VAR Clock,
                                           GNL_VAR *SourceClock, BLIST *PileOfInstCompo);

extern void GnlFreeClock (GNL_CLOCK *Clock);

extern GNL_STATUS GnlGetClocksFromNetwork (GNL_NETWORK Nw, BLIST
ListOfClocks,
                                         int PrintClkDomain);

extern GNL_STATUS TimeOptByResizingFromVar (GNL_VAR Var,
                                             LIBC_LIB Lib,
```

timeutil.e

```
float      Epsi,
float      MinSlack,
int       InoutIsIn,
BLIST     ListCritInst,
GNL      TopGnl,
BLIST     ListBuffers,
LIBC_CELL Buffer,
int       *Tag,
int       *Id,
int       IsAboutFanout);

extern GNL_STATUS TimeOptByResizingFromGnl (GNL Gnl, LIBC_LIB Lib,
                                             int IsAboutFanout,
                                             int *Tag, BLIST ListBuffers,
                                             LIBC_CELL Buffer);

/*-----*
/*-----* EOF -----*
```

703093 v1

00

12/28/00
JC992 U.S. PTO
09/752304

APPENDIX F

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

HUBBLE-RTL Overview

1 Introduction

HUBBLE-RTL is for now an internal tool and is under an alpha release. The goal of HUBBLE-RTL is to help the RTL-Designer to get Predicted information about the RTL design he is currently implementing. HUBBLE-RTL 'looks deeper in the RTL-design' by using state-of-art synthesis techniques and thus can predict accurate data. The following information is extracted:

- overall design information: Number of Flip-Flops, Latches, Tristates, Max Fanout, Max. Combinational Depth, ...,
- overall design gates area,
- overall design nets area,
- overall design critical paths (Non-linear delay model) and Epsilon Critical Region,
- technology verilog netlist.

All this information is related to a specific technology library which must be provided during the Synthesis/Estimation phase. Refer to the third and fourth sections to use HUBBLE-RTL in the *Synthesis/Estimation* mode. In the same time, HUBBLE-RTL provides an internal Netlist Checker which allows you to prove if two verilog netlists (built upon a technology library) are formally equivalent. Please refer to the section *Netlist Checker in HUBBLE-RTL* to use the verification mode capability. The tool runs on Polaris and linux platforms for now.

2 GENERAL FLOW WITH HUBBLE-RTL

2 General Flow with HUBBLE-RTL

HUBBLE-RTL can accept two inputs formats to describe a design:

- 'fsm' format which corresponds to the analysis and compilation of original VHDL descriptions thru VTIP-XL and FSMC (Tools used in VFormal). 'fsm' format is a kind of abstract netlist manipulating abstract objects (ex: state variables) and can be hierarchical (A hierarchical design is composed of several 'fsm' files).
- 'verilog' structural format dedicated only for netlists but not for RTL descriptions. Some 'verilog' constructs are not accepted like the UDPs for instances.

Moreover, HUBBLE-RTL needs a:

- technology library description in order to be able to predict some accurate information about the input design. For now, this technology library must be in the '.lib' format of 'Synopsys' but can come in the futur from the 'Milky Way' data-base.

HUBBLE-RTL accepts also several options which can tune several steps of the synthesis and give different reports. Refer to the next section and section *Options in HUBBLE-RTL* for more information.

As outputs, HUBBLE-RTL provides at the end of the excution :

- a Verilog netlist which is composed of the gates of the target library.
- Information related to this netlist used as prediction and thus accurate.

The general flow of HUBBLE-RTL can be sketch as follows:

00000000000000000000000000000000

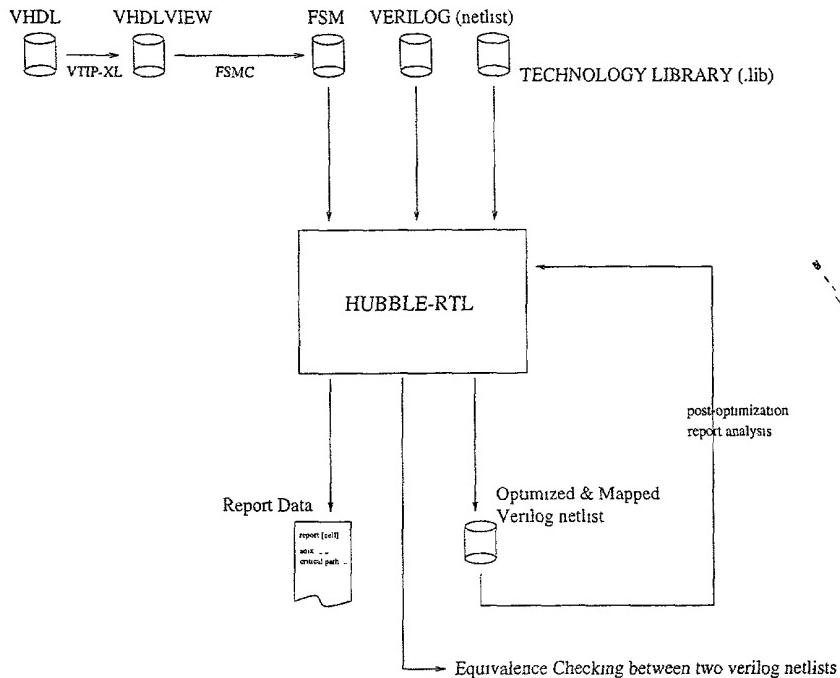


Figure 1: General flow in HUBBLE-RTL.

3 Optimizations in HUBBLE-RTL

HUBBLE-RTL performs both technology-independent optimizations and post-optimizations. Technology-independent optimizations can be invoked thru two options:

- -optimisation_strength [-os] [low—medium—high] : 3 levels of optimizations can be invoked in HUBBLE-RTL corresponding to *low*, *medium* and *high* effort. The *high* effort option calls some well-known synthesis technics to reduce the logic complexity of the design (2-level minimization, Algebraic factorization, ...) and thus can be longer in term of CPU time. On the other hand, the prediction must be much closer to any result given by a classical synthesis tool.
- -criterion [-c] [area_gate—area_net—timing—power] : 4 technics can be invoked during the technology mapping phase which is the first technology-dependant optimization. The option 'area_gate' will try to minimize the total gates area which is the sum of all the area of the gates constituting the netlist. 'area_net' tells the technology mapping phase to minimize the number of nets in the design so can reduce drastically the area due to nets. 'timing' tells the technology mapping to minimize the maximum depth in term of nets among the gates.

This criterion can be useful especially for sub-micronique technologies where net delays can be much more costly than gate delays. Finally 'power' will try to select the gates such that the sum of the probabilistic dynamic power of all of them is minimum.

Moreover, HUBBLE-RTL provides some Post-optimizations done after the technology mapping:

- Respect of library constraints: this is not purely a post-optimization since the goal here is to respect the constraints defined in the library file which concerns the max capacitance, max fanout allowed at the output of any gate. Meanwhile it brings generally a relative improvement of the design in term of critical path and epsilon critical region.
 - Gate resizing: HUBBLE-RTL plays on the gates having the same functionalities and choose the ones which will give the best result in term of critical path.

4 Information Report in HUBBLE-RTL

HUBBLE-RTL gives several reports about the current design. The type of report can be invoke with the option `-report_data [-rd] [none—modules—cells—timings]`. The report can be of 3 type:

- modules: in a hierarchical design, this report gives the respective areas of each module .

REPORT MODULES : DESIGN = [cat_top-structural]

- cells: this report gives the designer all the gates used in the design, the number of times they are used, the ratio they occupied in the total design, and the max fanout they drive.

REPORT CELLS : DESIGN = [crc_ck1-arch1]

Cell	Number	Cell_Area	Max_FanOut	Total_Area	% Design
NOR3_1K	1	1040.00	0	1040.00	0.1 %
XOR2_1K	12	15336.00	5	16376.00	1.8 %
NN2X2_1K	36	20808.00	1	37184.00	2.4 %
BUF1X2_1K	1	578.00	0	37762.00	0.1 %
XOR2X3_1K	4	5624.00	6	43386.00	0.7 %
OAI21_1K	175	121275.00	6	164661.00	14.2 %
DFFC_1K	10	27960.00	7	192621.00	3.3 %
DFFN_1K	2	5084.00	3	197705.00	0.6 %
MUX2_1K	37	47027.00	1	244732.00	5.5 %
AND2_1K	27	15606.00	5	260338.00	1.8 %
OR2X2_1K	2	1442.00	10	261780.00	0.2 %
BUF1_1K	9	4158.00	7	265938.00	0.5 %
INV1X20_1K	111	218115.00	4	484053.00	25.6 %
INV1_1K	39	13533.00	5	497586.00	1.6 %
AND5_1K	3	2772.00	5	500358.00	0.3 %
NN2D_1K	204	94248.00	5	594606.00	11.1 %
OAI21X6_1K	51	71400.00	7	666006.00	8.4 %
MUX2X3_1K	141	182031.00	1	848037.00	21.4 %
AND2X4_1K	1	931.00	0	848968.00	0.1 %
NN3D_1K	2	1156.00	1	850124.00	0.1 %
AND4X3_1K	1	1156.00	0	851280.00	0.1 %
Nb.Cells	869			851280.00	

- timings: this report gives the designer the critical path(s) occurring in the current design. It supports multiple clocks and will give critical path between each clocks. The critical path is computed according to the data defined in the library and with a non-linear delay model. This option can give also the size of the epsilon critical region of the design, e.g. the set of gates which have a slack less than an epsilon value specified by the user. This option also can print out the clock domain for each clock.

Performing Timing Analysis...

Report	:	timing
max_number_paths	:	1
Design	:	sm_arb-arch1
Date	:	07/07/99 @ 08:55:04
Operating Conditions	:	TYP
Library	:	CS80TYP_v5.0_t50
Wire Loading Model	:	TYPICAL

Point	Fanout	Incr	Path
input external delay oe_l (in)	0.0	0.0	

	0.000	0.000	Rising
m131(INV1_1K) (A-->Y \\$f14)	8	2.097	Falling
m134(AND2_1K) (B-->Y \\$f17)	2	1.441	3.538 Falling
m148(MUX2_1K) (D0-->Y \\$f18)	2	1.188	4.726 Falling
m153(MUX2_1K) (D0-->Y \\$f31)	1	1.011	5.737 Falling
smbgcpu(TINV2_1K) (A-->Y smbcpu)	1	0.420	6.156 Rising
smbgcpu (out)		0.000	6.156 Rising
Data Arrival Time			6.156

(Path is unconstrained)

Combinational Path 6.156 ns

Clock : clock35

Point	Fanout	Incr	Path
lastbg(0)(DFFBH_1K) (Q lastbg(0))	7	0.000	0.000 Rising
\\$IB_178(BUF1_1K) (A-->Y \\$IB_178)	4	1.535	1.535 Falling
m16(NND2X2_1K) (A-->Y \\$M_146)	1	0.750	2.285 Rising
m21(NND2X2_1K) (B-->Y \\$M_141)	1	0.455	2.741 Rising
m22(NND2X2_1K) (B-->Y \\$M_140)	1	0.346	3.086 Rising
m30(OAI21_1K) (B-->Y \\$M_132)	1	0.452	3.538 Falling
m133(MUX2X3_1K) (D1-->Y \\$f298)	2	1.325	4.864 Falling
m37(NND4_1K) (D-->Y \\$M_125)	1	0.632	5.496 Rising
m7(MUX2X3_1K) (D0-->Y \\$IM_171)	1	1.311	6.806 Falling
m137(INV1X20_1K) (A-->Y nextgnt(0))	2	0.277	7.083 Rising
m45(NOR2X3_1K) (B-->NOR2X3_1K \\$M_117)	1	0.778	7.861 Falling
m142(NND2X2_1K) (B-->Y \\$f58_47)	1	0.724	8.585 Rising
m159(INV1X20_1K) (A-->Y d_34)	1	0.289	8.874 Rising
bp0@grant(2)(DFFBH_1K) (D d_34)	0.0		8.874 Rising
Data Arrival Time			8.874

(Path is unconstrained)

Clock : clock35

Clock Period 8.87 ns

Clock Frequency

112.69 Mhz

5 Netlist Checker in HUBBLE-RTL

HUBBLE-RTL provides an utility command which reads two verilog netlists mapped on the same technology library (or using verilog primitives) and tells the user if both netlists are functionally equivalent. The user can define its own primary inputs mapping and select few primary outputs to verify. If errors are encountered, HUBBLE-RTL will provide counter-example under the form of boolean vectors. This command is quite usefull to verify any local improvment of the netlist (to check to different technology-independent optimization, different post-optimizations algorithm, customized modifications). A big assumption to use this command in the right way, is to have a one to one mapping between the registers of the two netlists. If this is the case, HUBBLE-RTL will try to map those registers automatically using the name of the output of the register. Note that register with Q and Q bar can be also handled in the netlist checker.

Below is an example on how we can invoke the netlist checker inside HUBBLE-RTL. This utility command is based on BDD technology and has been proven quite effective compared to industrial verification tool.

```
> hubble -mode verification -if vlg -lib synop.lib -i1 crc_low.v -i2 crc_high.v
#-----#
#
# HUBBLE-RTL version 9.2 Alpha - RTL Prediction & Synthesis
#
# Copyright (c) Avant! Corporation 1994-99. All rights reserved.
#
#-----#
#
Reading Verilog file [crc_low.v] ...
  o Analyzing module [crc_ck1-arch1]
Verilog Netlist Analyzed

Reading Verilog file [crc_high.v] ...
  o Analyzing module [crc_ck1-arch1]
Verilog Netlist Analyzed

WARNING: Top Level Module is [crc_ck1-arch1] by default
WARNING: Top Level Module is [crc_ck1-arch1] by default

Reading Library [synop.lib] ...
Library analyzed

Checking Hierarchical interfaces from top module [crc_ck1-arch1]

Performing Flattening of modules in [crc_ck1-arch1] (file='crc_low.v')
Flattening done

Performing Flattening of modules in [crc_ck1-arch1] (file='crc_high.v')
```

Flattening done

Linking Components With Libc Cells

Building Cell Functionalities..
DESIGN: [crc_ck1-arch1] (file='crc_low.v')

PIs	=	16
POs	=	12
PIOs	=	0
Dffs	=	12
Latches	=	0
Tristates	=	0

DESIGN: [crc_ck1-arch1] (file='crc_high.v')

PIs	=	16
POs	=	12
PIOs	=	0
Dffs	=	12
Latches	=	0
Tristates	=	0

Inputs Associations

```
-----
clk          <--> clk
reset        <--> reset
cnt[5]       <--> cnt[5]
cnt[4]       <--> cnt[4]
cnt[3]       <--> cnt[3]
cnt[2]       <--> cnt[2]
cnt[1]       <--> cnt[1]
cnt[0]       <--> cnt[0]
data[7]      <--> data[7]
data[6]      <--> data[6]
data[5]      <--> data[5]
data[4]      <--> data[4]
data[3]      <--> data[3]
data[2]      <--> data[2]
data[1]      <--> data[1]
data[0]      <--> data[0]
bp0@cx[0]   <--> bp0_cx[0]
bp0@cx[1]   <--> bp0_cx[1]
bp0@cx[2]   <--> bp0_cx[2]
bp0@cx[3]   <--> bp0_cx[3]
bp0@cx[4]   <--> bp0_cx[4]
bp0@cx[5]   <--> bp0_cx[5]
bp0@cx[6]   <--> bp0_cx[6]
bp0@cx[7]   <--> bp0_cx[7]
bp0@cx[8]   <--> bp0_cx[8]
bp0@cx[9]   <--> bp0_cx[9]
hecerror    <--> hecerror
pecrror     <--> pecrror
```

Outputs Associations

```
-----
hecerror    <--> hecerror
pecrror     <--> pecrror
q[0]         <--> q[0]
q[1]         <--> q[1]
q[2]         <--> q[2]
q[3]         <--> q[3]
```

```

q[4]          <--> q[4]
q[5]          <--> q[5]
q[6]          <--> q[6]
q[7]          <--> q[7]
q[8]          <--> q[8]
q[9]          <--> q[9]
nx[0]         <--> nx[0]
nx[1]         <--> nx[1]
nx[2]         <--> nx[2]
nx[3]         <--> nx[3]
nx[4]         <--> nx[4]
nx[5]         <--> nx[5]
nx[6]         <--> nx[6]
nx[7]         <--> nx[7]
nx[8]         <--> nx[8]
nx[9]         <--> nx[9]
d             <--> d
d_20          <--> d_20

```

Performing Equivalence Checking...

- o Verification started ...

- o Preprocessing ...

- o Structural checking ...

Nb Functions proved equivalent = 12

Nb Functions unproved = 12

Nb Bridge Nodes = 265

- o Local-BDD checking ...

Nb Functions proved equivalent = 24

Nb Functions proved non-equivalent = 0

Nb Functions unproved = 0

MaxNb Nodes = 500

Nb CutVar = 0

Nb Bridge Nodes = 1217

Nb InvBridge Nodes = 5

Output(s) proved different: 0

Output(s) not proved: 0

Verification done !

6 Performances of HUBBLE-RTL

HUBBLE-RTL has been tested on about 30 examples both with FSM or VERILOG inputs. It was run particularly on an industrial netlist of 500 Gates and here are the performances for each step of the synthesis:

- 1: Reading Verilog netlist (25 Megas): 32 secondes, 120 Megas.
- 2: Full flattening of the Verilog netlist: 1 minute, 340 Megas.
- 3: High optimization + Technology Mapping: 20 minutes.
- 4: Timing analysis: 2 minutes.
- The total time to read and print out the optimized verilog netlist is about 30 minutes. The improvement in term of gates area is 9%. Peak memory is 720 Megas.

7 Futur works in HUBBLE-RTL

HUBBLE-RTL is still in Alpha release and needs to be evaluated against a real synthesizer in the market to measure the gap of accuracy between the HUBLLE prediction and the synthesis tool result.

Some links must also be established in order to provide a uniform flow inside AVANT!. This means that the library solution must pass thru a tight integration with the Milky way data-base.

Finally, still some technical improvments must be adressed to raise the quality of the tool.

- Comparison with Synopsys DC starting from VHDL.
 - Evaluation of the FSMC inference.
 - Evaluation of the optimization phase.
 - Evaluation of the timing analysis accuracy.
 - Evaluation of the CPU/Memory comsumption vs. Synopsys.
- Integration with the 'Milky Way' data-base to get technology infos.
- Improvments of technology-indepent optimization.
- Implementation of Post-optimizations: pin permutations, local reconstruction of logic in epsilon critical region.
- Implementation of Power Analysis.
- Netlist Checker improvments:
 - Dynamic reordering implementation to avoid ordering sensitivity.
 - Strategy of cut var substitution in the False Negative Phase detection.
 - Improvement in the local BDD construction (Breadth instead of Depth search).

- Generation of the netlist representing the xor of the unproved outputs for post-simulation.

8 HUBBLE-RTL Options

```

#-----#
# HUBBLE-RTL version 9.2 Alpha - RTL Prediction & Synthesis #
# Copyright (c) Avant! Corporation 1994-99. All rights reserved. #
#-----#
Usage: HUBBLE-RTL 9.2 Alpha
-----
-mode [-m] [synthesis|estimation|translate|verification|post_optimization] :
  Specify if HUBBLE-RTL is in Synthesis, in Estimation, Translation,
  , Verification or Post-Optimization Mode.

-input [-i] <String> :
  Specifies file to use as input.

-output [-o] <String> :
  Specifies file to use as output.

-logfile [-log] <String> :
  Specifies file to use as log file; default is "stderr".

-input_format [-if] [fsm|vlg] :
  Specifies if the input format is .fsm (fsm) or verilog (vlg);
  default is 'fsm'

-fsm_dir [-fd] <String> :
  Specifies the path where all the FSM files of the current
  design are stored. Default is '.'

-optimisation_strength [-os] [low|medium|high] :
  Specifies strength with which logic optimization will be performed.

-criterion [-c] [area_gate|area_net|timing|power] :
  Specifies which criterion synthesis will optimize during Tech. mapping.
  Default is [area_gate]. This default option optimize the netlist in
  term of gates area. [area_net] minimizes the number of nets thus
  nets area. [timing] minimizes the longest path of nets between gates.
  Finally, [power] optimize the dynamic power consumption.

-vdd [-vdd] <Integer> :
  Specifies the voltage of the design in mV.

-use_sequential_qbar [-usq] :
  Specifies if output Q bar is used for sequential elements.

-max_cell_fanin [-mcf] <Integer> :
  Specifies the maximum FanIn of library cells taken into account.
  A greater max FanIn can slower the technology mapping phase but
  may give better results. By default the value is related to the
  option [-opt]. [-opt high] corresponds to the value 6 and otherwise

```

it will be 5. A maximum reasonable value you can use is 8.

-minimum_shared_logic_size [-msls] <Integer> :
 Specifies the minimum size of logic which will be shared during the optimization phase. Small size logic can influence badly the efficiency of the design because it involves generally a big FanOut. The size number is relative to the number of cubes composing the shared logic.

-use_verilog_primitives [-uvp] :
 Tells HUBBLE to print out an output netlist using Verilog primitives like 'or', 'and', 'buf', ... instead of using technology library cells. By default cells of the specified technology library are used.

-ignore_library_constraints [-ilc] :
 Tells HUBBLE to not take into account the library constraints (Max. capacitance, Max FanOut, Max Transition).

-ignore_post_optimization [-ipo] :
 Tells HUBBLE to not take into account the post optimization phase done on the mapped netlist. Post optimization performs netlist optimizations like buffer insertions, gates resizing, logic restructuring.

-print_clock_domain [-pcd] :
 Tells HUBBLE to print out the clock domains of each clock. This option is available only when -rd timings is invoked.

-print_epsilon_critical_region [-pecr] :
 Tells HUBBLE to print out the Epsilon critical region representing the number of critical gates compare to the total number of gates. This option is available only when -rd timings is invoked.

-lib [-l] <String> :
 Specifies the technology library (.lib format).

-inject [-inj] <Integer> :
 Specifies degree of logic reinjection performed. Greater is the integer and stronger is the flattening. Default is -1.

-flat_hierarchy [-fh] [none|all|except <List Strings>|only <List Strings>|leaf] :
 Specifies how the user hierarchy will be flattened. [none] means the hierarchy is preserved and [all] that it is completely flattened. [except] followed by a list of String specifies which module will not be flattened and [only] has the opposite meaning. [leaf] means that only the leaves modules will be flattened.

-flat_string [-fs] <String> :
 Specifies the String which separates instance names on a hierarchical path after flattening. By default the string is "..". Ex: U1.U2.U3 means that after the flattening, the current component comes from the instance path U1, U2 and U3. .

-flat_instance_name [-fin] [hierarchical|compact] :
 Specifies if the names of the flattened instances will correspond to the hierarchical name (ex: \inst1\inst2) or a compact name (name of the deepest instance) when flattening of the hierarchy is invoked.

-no_fold_identical_partition [-nfip] :
 Specifies if during the optimization phase, HUBBLE-RTL will try to fold

or not the identical partition of glue logic. This can significantly speed-up the logic optimization phase. By default the folding is performed but by using this option you disable the folding.

```

-top [-t] <String> :
    Specifies the top Verilog module from which the synthesis/estimation
    will be performed.

-y [-dir] <String> :
    Specifies the directory where leaf cell modules can be found.

+libext+ [-ext] <String> :
    Specifies the extension of the files defining the leaf cell modules.

-build_lib_force [-blf] [min|max] :
    Specifies if the .lib library will be used in a minimum or a
    maximum use. Maximum use is slower to build the library but
    results may be better because the complex cells may be instantiated.

-print_hierarchy [-ph] [none|before_flat|after_flat|both] :
    Invokes the printing of the design hierarchy. This can be done
    before the flattening phase (option [before_flat]) or after it
    (option [after_flat]) since the flattening can change the hierarchy.
    Option [both] prints out before and after flattening.

-report_data [-rd] [none|modules|cells|timings|powers] :
    Invokes the printing of a final report in which several data are
    presented related either to modules (use option [modules]) or to
    library cells (use option [cells]). Use option [timings] to get
    any information regarding timing issues in the design. Option
    [powers] will give average power and peak power in the design.

-print_nb_critical_path [-pncp] <Integer> :
    Defines the number of critical paths which will be printed out
    when the option [-rd timings] is invoked

-print_max_fanout_vars [-pmfv] :
    when the option [-rd cells] is invoked, this option forces HUBBLE-RTL
    to print out for each library cell, the corresponding instances
    with the maximum fan out

-input1 [-i1] <String> :
    Specifies the first Netlist used in the Netlist Checker.

-input2 [-i2] <String> :
    Specifies the second Netlist used in the Netlist Checker.

-read_name_association_file [-rnaf] <String> :
    Specifies the input file which defines the associations between the
    port names. The Netlist Checker will take into account these
    associations to prove the two netlists. If this option is not
    invoked then HUBBLE will try to figure out the associations by
    itself.

-print_name_association_file [-pnaf] <String> :
    Tells HUBBLE to print out the file of the ports associations used
    in the Netlist Checker. This file can be edited and re-read in HUBBLE
    with the option '-naif'. The ports associations can be then controlled
    in that way if you are not satisfied with the automatic ports
    association done in HUBBLE.

```

```
-verif_max_bdd_node [-vmaxbdd] <Integer> :  
    Specifies the maximum number of BDDs node before any break point  
    is introduced when the Equivalence Checker is invoked  
  
-ignore_random_simulation [-irs] :  
    tells HUBBLE not to call the random simulation phase during the Netlist  
    Checking.
```

9 Annexe 1: classical trace in HUBBLE_RTL

```
> hubble -i sm_arb-arch1.fsm -o toto.v -fd fsm -lib synop.lib

#-----
#  
# HUBBLE-RTL version 9.2 Alpha - RTL Prediction & Synthesis  
#  
# Copyright (c) Avant! Corporation 1994-99. All rights reserved.  
#  
#-----
```

Reading FSM file [fsm/sm_arb-arch1.fsm] ...
WARNING: Dead Code has been detected in file 'sm_arb.vhd'
 when entering line 271
WARNING: Dead Code has been detected in file 'sm_arb.vhd'
 when entering line 312
WARNING: Expression assigned to identifier grant MAY be OUT OF RANGE
WARNING: Expression assigned to identifier nextgt MAY be OUT OF RANGE

FSM File Analyzed [446]!

Signal	Type	Clock	Reset	Set	R/S
bp0@grant(0)	Dff	H	H	.	.
bp0@grant(1)	Dff	H	H	.	.
bp0@grant(2)	Dff	H	H	.	.
backoff	Dff	H	H	.	.
lastbackoff	Dff	H	H	.	.
lastbg(0)	Dff	H	H	.	.
lastbg(1)	Dff	H	H	.	.
lastbg(2)	Dff	H	H	.	.
lastmaster(0)	Dff	H	H	.	.
lastmaster(1)	Dff	H	H	.	.
lastmaster(2)	Dff	H	H	.	.
lastts	Dff	H	H	.	.
master(0)	Dff	H	H	.	.
master(1)	Dff	H	H	.	.
master(2)	Dff	H	H	.	.
nextmaster(0)	Dff	H	H	.	.
nextmaster(1)	Dff	H	H	.	.
nextmaster(2)	Dff	H	H	.	.

Nb. Dffs	18	18	0	0
Signal	Type	Select	Input	
smboff	TriState	H	H	
smbgsc3	TriState	H	L	
smbgsc2	TriState	H	L	
smbgsc1	TriState	H	L	
smbgsc0	TriState	H	L	
smbgcpu	TriState	H	L	

Nb. TriStates	6
---------------	---

Top Module(s):

- o [sm_arb-arch1]

WARNING: Top Level Module is [sm_arb-arch1] by default

Checking Hierachical interfaces from top module [sm_arb-arch1]

Reading Library [synop.lib] ...
Library analyzed

Building library elements...

Building Combinational Cell [157/157]

WARNING: cell is ignored because of complex next state function: [JKBX6_1K]
 WARNING: cell is ignored because of complex next state function: [JKBX3_1K]
 WARNING: cell is ignored because of complex next state function: [JKBX2_1K]
 WARNING: cell is ignored because of complex next state function: [JKB_1K]
 WARNING: cell is ignored because of complex Input pin function: [TZMUX2_1K]
 WARNING: cell is ignored because of complex Input pin function: [TZMUX2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [TZMUX2X3_1K]
 WARNING: cell is ignored because of complex Input pin function: [TMUX2_1K]
 WARNING: cell is ignored because of complex Input pin function: [TMUX2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [TMUX2X3_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTZMU2_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTZMU2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTZMU2X3_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTMUX2_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTMUX2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [NTMUX2X3_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDL_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDL2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDL2X3_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDH_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDH2X6_1K]
 WARNING: cell is ignored because of complex Input pin function: [IDH2X3_1K]

Library Feature:

- o NB. DFFS = 20
- o NB. LATCHES = 15
- o NB. 3-STATES = 12
- o NB. BUFFERS = 4

=====

SYNTHEZISING [sm_arb-arch1]

```
=====
  o NB. INPUTS      = 9
  o NB. OUTPUTS    = 6
  o NB. INOUTS     = 0
  o NB. LITERALS   = 454
  o NB. FLIP-FLOPS = 18
  o NB. LATCHES    = 0
  o NB. TRISTATES  = 6
  o NB. BUFFERS    = 0
  o MAX. COMBINATORIAL PATH = 27
```

```
Constant Pushing in [sm_arb-arch1]...
Collapsing glue logic in [sm_arb-arch1]...
Selective re-injection of logic in [sm_arb-arch1](Inject=-1)...
  o LITERALS      = 293
  o MAX. COMB. PATH = 27
Constant Pushing in [sm_arb-arch1]...
Reinjecting glue logic (size = 2) [72/72]
Sorting functions...
Structuring (1) for mapping...
Structuring (2) for mapping...
```

```
Technology Mapping:
  o Criterion = AREA
  o Mapping Equation [64/64]
Sorting Gates...
Building cell [64/64]
Gates compaction...
Post mapping optimization [166/166]
```

```
Extracted Data After Technology Mapping:
  o Total Gates Area      = 198336 units
  o Max. Comb. Wire Path  = 15 wires
  o Max. Fanout           = 18 nets
```

```
-----
Report      : area
Design      : sm_arb-arch1
Version     : 1999.01
Date        : 07/07/99 @ 08:57:30
Library Used : CS80TYP_v5.0_t50

Number Of Ports      : 15
Number Of Cells      : 189
Combinational Area   : 104839.00
Non Combinational Area: 93497.00
Net Interconnect Area : 153396.00
Total Area           : 351732.00
```

```
Restructuring Netlist to meet Library Constraints...
```

```
-----
Report      : area
Design      : sm_arb-arch1
Version     : 1999.01
Date        : 07/07/99 @ 08:57:30
Library Used : CS80TYP_v5.0_t50
```

```
Number Of Ports      :      15
Number Of Cells     :      198
Combinational Area  : 109461.00
Non Combinational Area: 93497.00
Net Interconnect Area : 156394.50
Total Area          : 359352.50
```

Performing Post-Optimization...

Optimization By Resizing Critical Instances...

```
-----  
Report      : area
Design       : sm_arb-arch1
Version      : 1999.01
Date         : 07/07/99 @ 08:57:30
Library Used : CS80TYP_v5.0_t50
```

```
Number Of Ports      :      15
Number Of Cells     :      198
Combinational Area  : 138673.00
Non Combinational Area: 93497.00
Net Interconnect Area : 156394.50
Total Area          : 388564.50
```

```
=====  
GLOBAL NETLIST REPORT  
=====  
o TOP-LEVEL           = sm_arb-arch1  
o TECH LIBRARY        = CS80TYP_v5.0_t50  
o GATES AREA          = 232170.00  
o NETS AREA            = 156394.50  
o TOTAL NETLIST AREA  = 388564.50  
o NB. EQUIVALENT GATES = 401.68  
o NB. FLIP-FLOPS       = 18  
o NB. LATCHES          = 0  
o NB. TRISTATES        = 6  
o NB. BUFFERS          = 0  
=====
```

Printing Verilog Netlist 'toto.v'...

Synthesis Done !



APPENDIX G

IMPLICIT MAPPING OF TECHNOLOGY INDEPENDENT NETWORK TO LIBRARY CELLS

THIERRY BESSON

M-7928 US

Implicit Pattern Matching

T.Besson

Abstract

This paper addresses the general problem of technology mapping of a Boolean network using a cells library. It presents an original solution to find directly all the matches of a given Boolean function f without resolving hashing collision or performing several boolean function equivalences. The implicit matching is done through a ROBDD construction of f whose size cannot exceed $2^M/M$ where M is the max fan in of the cells library. This is relatively different from classical methods which can be time consuming because a minimum number of boolean comparisons is required to find all the matches. Here, the complexity of the problem is exported from the matching/scanning cells phase to the organization and construction of the cells library. This approach has been implemented in C in an industrial tool which performs quick synthesis. Some results presented at the end demonstrate the important impact of the implicit pattern matching solution.

1 Introduction

Given a Boolean network representing a multi-level combinational logic circuit and a cells library, the task of *technology mapping* is to bind nodes in the network to cells in the library [9]. Classical methods - either structural-based (tree matching/graph matching [9]) or functional-based (Boolean matching) [4][7] - use a common global approach. A given function f of the Boolean network is identified and then compared against a set of cells $\{c_i\}_i$. The most popular approach to area minimization is to reduce technology mapping to network covering and approximate this one by a sequence of tree coverings which can be performed optimally by dynamic programming [9]. However, this sequence of tree coverings can be time consuming if the library size is important since each sub-tree of the network is compared to the tree representation of each cell. In order to decrease this complexity, some have tried either to minimize the cost of the matching function and/or to minimized the number of comparisons. A classical solution to this consists to define signatures or keys for each cells in order to reduce the overall complexity. A complete survey has been done by [1] which addressed all these methods. For instance [12] introduced keys to hash the library elements in order to avoid time-consuming one-to-one function equivalence. Very recently, [8] introduces the canonical key $R[f]$ identifying a NPN equivalence class. Despite the fact that the canonical representative $R[f]$ can drastically reduce the number of comparisons, its size representation is 2^n (if n is the support size of f) and its computation can be costly even by using some properties. In the sequel, we present a new approach called *Implicit Pattern Matching* which, in comparison with the signature or hash key based methods, does not need to resolve hashing collision or to perform several boolean function equivalences which is usually computationally expensive.

Chapter 2, gives quickly the formulation of the problem we refer to and conventions. Chapter 3 explained in details the *Implicit Pattern Matching* and represents the core of the paper. Chapter 4 explains how the *Implicit Pattern Matching* can be embedded in a classical dynamic-programming based technology mapping. Chapter 5 gives experimental results and in chapter 6 a conclusion about future works is given.

2 Problem Formulation

We consider in the sequel Boolean functions that model a portion of the circuit and that are called *target functions*. We denote by f a general target function. We call *pattern function* a combinational function modeling a library cell c_i , and we use g_{c_i} to represent the pattern function associated to cell c_i . We assume moreover that both target and pattern functions have a single output.

To implement the *Implicit Pattern Matching* presented in the next chapter, we use the notion of Typed ROBDD that are explained for instance in [11].

The general problem addressed in this paper is then the following. Let f be a subcircuit's function and $L = \{c_1, \dots, c_n\}$ a target library. The goal is to know the subset of cells c_i which can match the function f . This means that this subset can be empty (no cell can match f), can have a single element (then only one cell can realize f) or several elements (then a 'best' cell can be selected among these elements). The problem will address also configured cells which correspond to original cells where some inputs have been fixed to constant values and/or bridged to other inputs.

3 Implicit Pattern Matching

The goal of the *implicit pattern matching* is to find all the pattern functions g_{c_i} of cells c_i which match a given function f without performing any library scan. To do so, we need to introduce several definitions to present a general formulation of the *implicit pattern matching*.

3.1 Formulation

Definition 3.1.0:

A *variable assignment* from a Boolean vector $X = \{x_1, \dots, x_n\}$ to a Boolean vector $Y = \{y_1, \dots, y_j, \dots, y_n\}$ is a bijection \mathcal{A}_k between X and Y where for each index i, j we have $y_j = \mathcal{A}_k(x_i)$ and $x_i = \mathcal{A}_k^{-1}(y_j)$. We note $Y = \mathcal{A}_k(X)$ (resp. $X = \mathcal{A}_k^{-1}(Y)$) the assignment from X to Y (resp. from Y to X). We note $\Lambda(X, Y)$ the set of all the possible assignments $(\mathcal{A}_k)_k$ from X to Y .

Definition 3.1.1:

Let f be a Boolean function and X the support of variables

of f (we note f by $f(X)$). The ROBDD built on the function $f(Y = \mathcal{A}_k(X))$ with the order $\pi = (y_1 \prec \dots \prec y_i \prec \dots \prec y_n)$ is called *Canonical Representant* of f upon vector Y for variable assignment \mathcal{A}_k and is noted $\mathcal{R}_{Y=\mathcal{A}_k(X)}(f)$.

Definition 3.1.2:

The *Canonical Representants Set* of f upon a set of variables $Y = \{y_1, \dots, y_n\}$, noted $\mathcal{CRS}_{XY}(f)$, is the set of *Canonical Representant* $\mathcal{R}_{Y=\mathcal{A}_k(X)}(f)$ for all $\mathcal{A}_k \in \Lambda(X, Y)$.

Example:

Let consider the function $f = x_1.x_2 + x_3$. The $\mathcal{CRS}_{XY}(f)$ has three elements described in fig 1. ROBDD (1) corresponds to variable assignments $(y_1 = \mathcal{A}_1(x_1), y_2 = \mathcal{A}_1(x_2), y_3 = \mathcal{A}_1(x_3))$ and $(y_1 = \mathcal{A}_2(x_2), y_2 = \mathcal{A}_2(x_1), y_3 = \mathcal{A}_2(x_3))$. ROBDD (2) corresponds to variable assignments $(y_1 = \mathcal{A}_3(x_1), y_2 = \mathcal{A}_3(x_3), y_3 = \mathcal{A}_3(x_2))$ and $(y_1 = \mathcal{A}_4(x_2), y_2 = \mathcal{A}_4(x_3), y_3 = \mathcal{A}_4(x_1))$. Finally, ROBDD (3) corresponds to variable assignments $(y_1 = \mathcal{A}_5(x_3), y_2 = \mathcal{A}_5(x_1), y_3 = \mathcal{A}_5(x_2))$ and $(y_1 = \mathcal{A}_6(x_3), y_2 = \mathcal{A}_6(x_2), y_3 = \mathcal{A}_6(x_1))$.

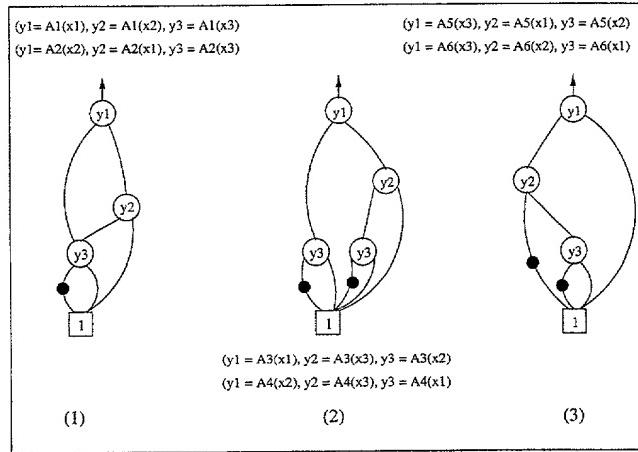


Figure 1: Canonical Representants Set of f

The size of $\mathcal{CRS}_{XY}(f)$ is 3 and not !3 because the function has two symmetric variables which involves same ROBDDs for the assignments presented above.

Proposition 3.1.3:

If there exists a variable assignment \mathcal{A}_i from X to X' which makes a function $f'(X')$ functionaly equivalent to a function $g(X)$ then for all variable assignment $\mathcal{A}_k \in \Lambda(X', Y)$, the ROBDD

$\mathcal{R}_{Y=\mathcal{A}_k(X')}(f')$ belongs to $\mathcal{CRS}_{XY}(g)$ and conversely. We note:

$$(\exists \mathcal{A}_i, f'(X') = g(\mathcal{A}_i(X)) \Leftrightarrow (\forall \mathcal{A}_k, \mathcal{R}_{Y=\mathcal{A}_k(X')}(f') \in \mathcal{CRS}_{XY}(g))) \quad (1)$$

Moreover, this involves that the *Canonical Representants Set* of f' and g are exactly the same, we note:

$$(\exists \mathcal{A}_i, f'(X') = g(\mathcal{A}_i(X)) \Leftrightarrow (\mathcal{CRS}_{X'Y}(f') = \mathcal{CRS}_{XY}(g))) \quad (2)$$

The previous proposition uses a fundamental property on ROBDDs. This property specifies that if two ROBDDs are functionaly equivalent then they have a same unique ROBDD representant (cf. [3]).

Let us choose a variable assignment \mathcal{A}_k . Since $\mathcal{R}_{Y=\mathcal{A}_k(X')}(f') \in \mathcal{CRS}_{XY}(g)$ then there exists a \mathcal{A}_l such that $ROBBD_{Y=\mathcal{A}_k(X')}(f') = ROBBD_{Y=\mathcal{A}_l(X)}(g)$. Consequently the ordering $(y_1 \prec \dots \prec y_i \prec \dots \prec y_n)$ with assignments $Y = \mathcal{A}_k(X')$ or $Y = \mathcal{A}_l(X)$ gives the same ROBDD. This means that there exists a relation between X' and X which verifies:

$$(Y = \mathcal{A}_k(X') = \mathcal{A}_l(X)) \Leftrightarrow (X' = \mathcal{A}_k^{-1}(\mathcal{A}_l(X)) = \mathcal{A}_i(X)) \quad (3)$$

$$\Leftrightarrow \mathcal{A}_i = \mathcal{A}_k^{-1}(\mathcal{A}_l) \quad (4)$$

Proposition 3.1.3 has two interpretations corresponding to equations (1) and (2). First of all, if the ROBDD of a function f' with a variable assignment \mathcal{A}_k is in a set $\mathcal{CRS}_{XY}(g)$ then f' and g are functionaly equivalent with the assignment \mathcal{A}_i defined by (4). This means that the functional equivalence between a Boolean function f' and a pattern function g comes down to build a single ROBDD of f' with a variable assignment $Y = \mathcal{A}_k(X')$. The second meaning is that if f' and g are two equivalent pattern functions then they have exactly the same *Canonical Representants Set* according to (2). This strong proposition is the basis of the Implicit Pattern Matching since we show here that a single ROBDD construction of a function f' can decide if f' is functionally equivalent to a set of pattern function g_i having all the same *Canonical Representants Set*.

This is not the case of classical approaches which generally need to perform at the end a Boolean equivalence (through isomorphism between two Bdds [10], through Boolean unification between two Bdds [5], through truth table equivalence [8], through string comparisons [13], ...) despite the fact that they used signatures and hash keys to reduce the number of function-to-function comparisons.

Definition 3.1.4: We define the *Canonical Representants Set* of a library cells L upon a vector Y , noted $\mathcal{CRS}_Y(L)$, by the following relation:

$$\mathcal{CRS}_Y(L) = \bigcup_{c_i \in L} \mathcal{CRS}_{X_i Y}(g_{c_i}) \quad (5)$$

Proposition 3.1.5:

A Boolean function $f'(X')$ has matches in the library cells L if for any variable assignment \mathcal{A}_k the ROBDD $\mathcal{R}_{Y=\mathcal{A}_k(X')}(f')$ belongs to $\mathcal{CRS}_Y(L)$. Then matching cells correspond to the set $Cells(f'(X'))$ defined by (7).

$$(f' \text{ has a match in } L) \Leftrightarrow (\forall \mathcal{A}_k, \mathcal{R}_{Y=\mathcal{A}_k(X')}(f') \in \mathcal{CRS}_Y(L)) \quad (6)$$

$$Cells(f'(X')) = \{c_i \in L / \forall \mathcal{A}_k, \mathcal{R}_{Y=\mathcal{A}_k(X')}(f') \in \mathcal{CRS}_{X_i Y}(g_{c_i})\} \quad (7)$$

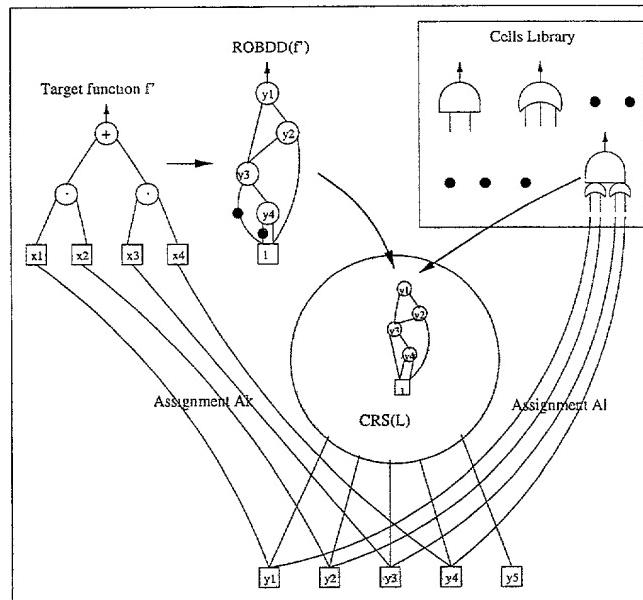


Figure 2: General approach in Implicit Pattern Matching

3.2 Reducing the *Canonical Representant Set* of a library

We formulated a way to find all the matches of a given Boolean function f by just constructing a single ROBDD of f . The complexity of the problem is no more on the matching phase

but on the way to represent *Canonical Representant Sets* which can be huge for cells with a lot of inputs. The size of a *Canonical Representants Set* can be reduced if one can reduce the possible assignments allowed. By considering canonical signatures on each x_i some possible assignments can be eliminated. Proposition 3.2.0 presents a way to do it.

Proposition 3.2.0:

Let $S(x_i)$ an integer which is a canonical signature of variable x_i in function $f(X)$. Then $\mathcal{CRS}_{XY}(f)$ can be restricted to ROBBDs $\mathcal{R}_{Y=\mathcal{A}_k(X)}(f)$ such that \mathcal{A}_k verifies:

$$\forall m, n \quad (y_m \prec y_n) \Rightarrow (S(\mathcal{A}_k^{-1}(y_m)) \leq S(\mathcal{A}_k^{-1}(y_n))) \quad (8)$$

\mathcal{A}_k is said *compatible with respect to signature S* if the equation above is verified and in the sequel such \mathcal{A}_k will be referenced as a *compatible assignment*.

The signature $S(x_i)$ used here corresponds to the number of minterms of the function f_{x_i} , e.g. the cofactor of f with respect to variable x_i . This signature can be efficiently computed on a ROBDD representation of f and is linear in term of the number of Bdd nodes. In the previous example the following signatures are extracted: $S(x_1) = 3$, $S(x_2) = 3$, $S(x_3) = 4$. The set of possible assignments for f are then $(y_1 = \mathcal{A}_1(x_1), y_2 = \mathcal{A}_1(x_2), y_3 = \mathcal{A}_1(x_3))$ and $(y_1 = \mathcal{A}_2(x_2), y_2 = \mathcal{A}_2(x_1), y_3 = \mathcal{A}_2(x_3))$ and the other assignments are eliminated because they are not compatible with signature S . Then $\mathcal{CRS}_{XY}(f)$ has one single element which is the ROBDD (1) of fig. 1.

For instance, the different sizes of the $\mathcal{CRS}_{XY}(g_{c_i})$ for all the cells c_i of the MCNC library *lib2.lib* are presented in the table 1.

Cells	Size	Cells	Size	Cells	Size
inv1x	1	inv2x	1	inv4x	1
xor	1	xnor	1	nand2	1
nand3	1	nand4	1	nor2	1
nor3	1	nor4	1	aoi21	1
aoi31	1	aoi22	3	aoi32	1
aoi33	10	aoi211	1	aoi221	3
aoi222	15	oai21	1	oai31	1
oai22	3	oai32	1	oai33	10
oai211	1	oai221	3	oai222	15

Table 1: Size of the Canonical Representants Sets of *lib2.lib* cells

We note that the size of $\mathcal{CRS}_{XY}(g_{c_i})$ is relatively reduced by using only compatible assignments with signature S^1 . By considering only *compatible assignments* when constructing ROBDDs for both pattern functions and target functions we still respect all the propositions presented in section 3.1. Several others methods can be used to reduce drastically these sets but all must still respect the fundamental propositions enumerated in the previous section.

3.3 Implementing the *Implicit Pattern Matching*

3.3.1 The library representation

In order to build an efficient library representation in memory it is useful to store for each ROBDD $\mathcal{R}_A(g_{c_i})$ the corresponding cell c_i . Since this ROBDD may have been constructed from several cells, then it can refer to all of these cells. Fig 2. shows a possible implementation of some library cells from the MCNC library *lib2.lib*.

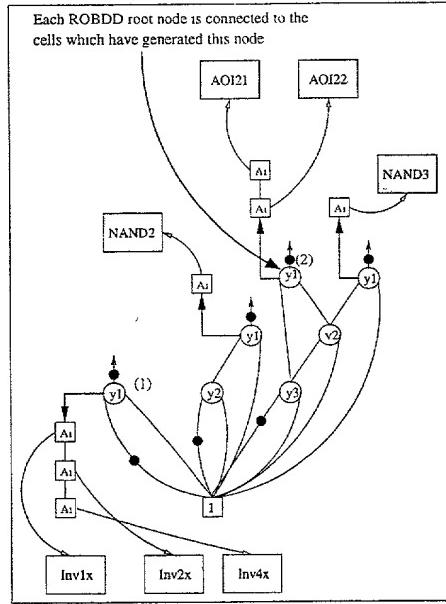


Figure 3: Implementation of the cells library with ROBDDs

For each ROBDD root node there is a list of cells whose functions correspond to this ROBDD. For each pointed function c_i , we

¹For instance the size $\mathcal{CRS}_{XY}(g_{c_i})$ for $c_i = AOIx_1x_2\dots x_n$ is in general $C_n = (!\sum_{i=1}^n x_i)/(!n \cdot \prod_{j=1}^n !x_j)$. If for instance x_n is less than the other x_i , then applying compatibility according to signature S reduces the size of C_n to size of C_{n-1} . For instance size of *aoi221* is size of *aoi22*.

store \mathcal{A}_i which allowed to get $Y = \mathcal{A}_i(X)$ where X is the support of g_{c_i} . For instance the root node (1) points on 3 cells $Inv1x$, $Inv2x$ and $Inv4x$. The root node (2) points on two cells $AOI21$ and configured cell $AOI22$ (exactly: $AOI22(a1, a2, b1, 1)$). Since this information must be accessible at the Bdd node level, the Bdd node structure must be extended in order to store it.

3.3.2 ROBDD construction of a target function f'

According to *Proposition 3.1.5*, the ROBDD construction of a target function f' with a compatible assignment \mathcal{A}_k can belong or not to $\mathcal{CRS}(L)$, if L is the target library. Given a compatible assignment \mathcal{A}_k and a function f' corresponding to the node (1) in figure 4, then $f' = !(a.b + c)$, a compatible assignment can be $(y_1 = \mathcal{A}_1(a), y_2 = \mathcal{A}_1(b), y_3 = \mathcal{A}_1(c))$ or $Y = \mathcal{A}_1(X')$ where $X' = \{a, b, c\}$. Then the ROBDD $\mathcal{R}_{Y=\mathcal{A}_1(X')}(f')$ has a root node which is exactly the node (2) of Fig. 3 or 4. Moreover, from this root node we inherit directly and implicitly of all the cells which can realize the function f' since we can access all the cells from bdd node (2). Note that the cells found can realize the function or its complement because of the Types ROBDD representation². Finally, to know the variable assignment between f' and a gate c_i we need to compose (as defined in equation (4)) \mathcal{A}_1 with the variable assignment \mathcal{A}_i pointing on c_i .

In term of complexity, the *implicit pattern matching* for a function f against a library cells L needs only to identify a compatible assignment variable \mathcal{A}_k (e.g computing signature for each variable x_i of the support of f which can be done by computing a Bdd of f) plus the construction of ROBDD $\mathcal{R}_{Y=\mathcal{A}_k(X)}(f)$. In the worst case, two ROBDDs constructions are necessary to find all the cells c_i which match f but one ROBDD construction can be enough because signatures can be also computed on $\mathcal{R}_{Y=\mathcal{A}_I(X)}(f)$ itself where for each i , $y_i = \mathcal{A}_I(x_i)$. If by "chance" \mathcal{A}_I is compatible with S then one construction is enough. Practically this is the case when function has a lot of symmetric variables (ex: Nand2, Nand3, ... needs only 1 ROBDD construction). If the max fan in of the library L is n , then the two ROBDDs can have a maximum size of $2^n/n$ bdd nodes [6].

²Generally speaking, NPN function solutions are treated in this way: Output Inversion with typed ROBDD structure, Input permutation with Canonical Representant Set, Input inversions with double-inverters insertions explained later on.

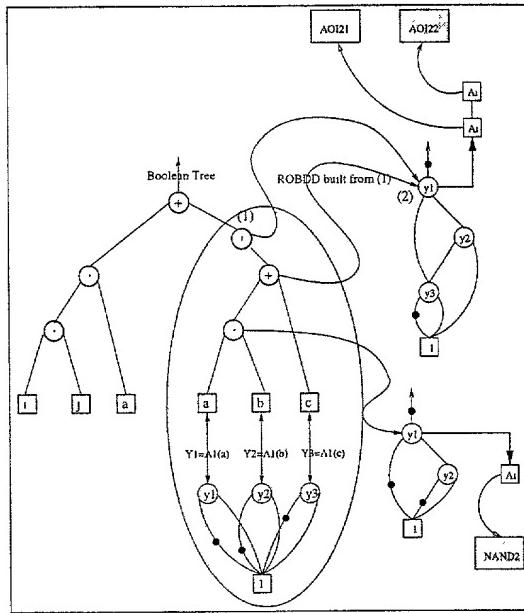


Figure 4: Implicit pattern matching on a Boolean tree

In other matching algorithms, several objects must be constructed to perform the matching. In [10], ROBDD for the pattern function and for each target function (called *cluster function*) must be built. Moreover ROBDD isomorphisms are performed to isolate a match which is also time consuming. In [5] Boolean unification is used in order to unify two boolean function f and g by finding their *mgu* (Most general unifier). Again this is done for all the pattern functions and the *mgu* computation can be costly since it performs a *Shannon* decomposition. Methods with signatures or hash keys, despite eliminating a lot of pattern functions, still need to perform a final functional matching in order to resolve the problem of *aliasing*. This problem corresponds to the situation where a *bucket* of elements have the same keys but can correspond to different functionalities. It is then necessary to functionally check these elements in order to extract the correct ones. Coupling the *implicit pattern matching* and the Signature computation needs in the worst case two ROBDD constructions.

4 Technology mapping with *Implicit Pattern Matching*

The technology mapping described here use a classical dynamic programming approach starting recursively from the root node of the Boolean tree down to the leaves which can be either primary inputs or sequential element outputs. Thus, the optimal mapping at node n is defined by the equation below:

$$MapOpt(n) = MIN_{c_i \in L}(Cost(c_i) + \sum_{n_j \in Fanin(Cov(c_i))} (MapOpt(n_j))) \quad (9)$$

where c_i are the cells matching at node n , $Cost(c_i)$ is the cost of the cell c_i , $Fanin(Cov(c_i))$ the inputs of the cover of node n by cell c_i . In the sequel, we present a general algorithm *MapOptArea* and *MapBestCellArea* for Area optimization which mixed the *Implicit Pattern Matching* procedure and the dynamic-programming based algorithm. The algorithm is called on a node 'N' of the Boolean network and the best area information is stored on this node through the field 'GetNodeArea'.

```

MapOptArea (N)
begin
    /* Node tree is a constant or a primary variable.
    if ((N = CONSTANT) or (N = VARIABLE))
    {
        SetNodeArea (N, 0);
        return;
    }
    if (GetNodeArea (N) <> INFINITE)      /* Already computed.
        return;

    /* First support is the list of the Sons of 'N'.
    Support <- Sons (N);

    MapBestCellArea (N, Support, 0);
end

MapBestCellArea (N, Support, Limit)
begin
    /* Performing recursively support expansion for index 'i' >= 'Limit' */
    for (i<-0; i<Size(Support); i++)
    {
        SonI <- Support[i];
        if ((i >= Limit) and
            (N <> CONSTANTE) and
            (N <> VARIABLE))
        {
            /* 'Support' is expanded into 'NewSupport' where 'SonI'
            /* is replaced by its sons.
            /*

```

```

NewSupport <- Support{SonI <- Sons (SonI)};
if (Size (NewSupport) > MaxFanIn)
    continue;

    MapBestCellArea (SonI, NewSupport, i);
}
}

/* Each node 'SonI' assigned to a library input variable. */
for (i<-0; i<Size(Support); i++)
    Assign (SonI, LibraryInput[i]);

/* ROBDD construction corresponding to the Implicit Pattern Matching. */
/* Leaves during ROBDD construction corresponds to previously assigned*/
/* Nodes 'SonI'. */
bdd <- build_robdd (N);

if (Cells (bdd))
{
    Area <- BestCellArea (Cells (bdd)); /* Extract the best area among*/
                                         /* the Cells. */
    for (i<-0; i<Size(Support); i++)
    {
        SonI <- Support[i];

        /* Calling recursively the best area mapping. */
        MapOptArea (SonI);

        Area <- Area + GetNodeArea (SonI);

        /* if actually there is already a better solution. */
        if (Area >= GetNodeArea (N))
            break;
    }
    if (Area < GetNodeArea (N))
    {
        SetNodeArea (N, Area);
    }
}
end

```

5 Experimental Results

The *implicit pattern matching* has been implemented in an industrial tool (internal name *MAPIT*) which intends to do quick and efficient synthesis. This tool, including the *implicit pattern matching* engine and the BDD package, has been written in C by the author. Experimental tests have been performed on a HP 735/125 in order to do direct comparisons with [8]. These results give the optimum solution in term of Area when mapping MCNC benchmarks with library *lib2.lib*. Indeed, for each node all the matching cells are identified by the implicit pattern matching procedure and then a recursive dynamic programming approach is used to get the optimum. Two options are run which are

referenced as *MAPIT D* and *MAPIT I*. *MAPIT D* starts from an original netlist where single local variables are collapsed and where the general Boolean tree is transformed into a classical 2-AND/INV tree. *MAPIT I* starts almost from the same 2-AND/INV Boolean tree except that for each edge between two nodes a double inverter INV is inserted. This insertion is useful to capture input polarities. This solution must be necessarily better than *MAPIT D* because the starting Boolean tree description has been refined. Table 2 gives the number of target functions f considered when invoking the algorithm presented previously for both *MAPIT D* and *MAPIT I*, the number of constructed ROBDDs and the number of matching cells.

Benchs	MAPIT D			MAPIT I		
	#f	#ROBDDs	#Cells	#f	#ROBDDs	#Cells
c1355	24726	45856	3858	1801232	3560718	48708
c1908	16590	30356	2995	630820	1239353	24431
c2670	39121	73949	4619	1784184	3534214	37003
c3540	71128	135692	7210	4769306	9470356	78933
c499	10878	19640	2314	189648	360430	21220
c5315	93290	176712	10810	3747627	7413442	92319
c6288	48013	84833	12183	491917	892960	98901
c7552	124219	235682	13765	4053312	8006523	113591
alu4	87426	170465	4895	4317346	8595754	46811
apex6	40550	77207	4332	1807701	3581550	40793
des	314324	602009	28024	15087856	29926765	268264
frg2	90438	172929	8523	3114734	6171145	64652
k2	95607	179658	11616	6286366	12442418	131265
pair	110526	212169	9818	6163316	12240985	104682
rot	36670	70068	3574	1265059	2504463	30089
too_large	87793	170341	5600	3248326	6451654	51005
vda	48501	91271	5732	2306731	4558462	55001
x3	54452	103761	5646	1655113	3273201	42650

Table 2: Complexity study of *Implicit Pattern Matching* on C* examples.

In Table 2, the number of ROBDD constructions is about 1.9 times the numbers of target functions which is in practice the general cost to perform all the matches. Moreover the number of target functions increased a lot between option *MAPIT D* and *MAPIT I* but then the number of cells founds is much more important which leads to a larger exploration space.

Table 3 presents results comparisons between *MAPIT* and those published by [8] which includes results on *SIS 1.3* [2]. Note that all area values have been scaled down by 464 (the greatest common divisor) for a sake of clearness. The *MAPIT D* option clearly shows an important speed-up in term of CPU time and area improvement compared to *TEMPLATE* and *SIS 1.3*. The

speed-up is about 38 times faster (resp. 114 times) than *SIS 1.3* (resp. *TEMPLATE*) and the gain in term of Area is about 11.1 % (resp. 3%). Regarding *MAPIT I* option the cpu-time is the same as *SIS 1.3*, and 3 times faster than *TEMPLATE*. The area gain is about 19.2 % better than *SIS 1.3* and 11.1 % better than *TEMPLATE*. Note that sometimes *MAPIT I* can drastically improved results compared to *MAPIT D* such as example *c6288*.

Benchs	TEMPLATE		SIS 1.3		MAPIT D		MAPIT I	
	Area	CPU	Area	CPU	Area	CPU	Area	CPU
c1355	870	270	1748	48	826	1.7	826	170.3
c1908	1330	322	1640	62	1183	1.3	1134	53.1
c2670	1965	687	2145	116	1841	3.1	1840	155.3
c3540	2808	942	3020	141	2833	6.1	2671	428.0
c499	870	228	1172	39	826	0.7	826	12.7
c5315	4420	1864	4546	379	4482	10.1	4216	325.0
c6288	6264	186	7910	286	6877	9.2	5486	32.1
c7552	5631	1697	6917	573	5518	18.9	5264	368.4
alu4	1710	743	1753	80	1748	7.2	1675	404.6
apex6	1637	466	1623	67	1664	3.4	1580	164.6
des	11909	4388	12142	2833	10628	55.8	9542	1407.9
frg2	3529	1144	3235	236	3177	7.4	2913	268.7
k2	5902	1590	6247	534	5316	10.5	4795	526.5
pair	3601	1090	3891	198	3809	10.2	3664	568.3
rot	1594	449	1540	71	1621	2.9	1506	114.7
too.large	2250	904	2172	183	1926	6.8	1855	293.5
vda	2818	691	3123	154	2531	3.6	2325	187.3
x3	2376	751	2100	214	2295	4.2	2021	140.0
TOTAL	61484	18412	66924	6214	59101	163.1	54049	5621.0
GAIN/SIS	8.1%	-3x	-	-	11.1%	+38x	19.2%	+1.1x

Table 3: Results for area optimization.

The results given by *MAPIT I* are 'a priori' the optimum but it is not the case for circuit *pair* since *TEMPLATE* is better. This must come from the fact that the original Boolean networks are probably not the same.

Table 4 represents the two options where both original gates and configured gates are used in *MAPIT I*. In the configured case, results are not much improved because with this library most of the configured cells can be built with original library cells with less cost. Note that the increase of the number of considered cells between the two cases has almost no impact on the CPU time. Only the time to build the library is a little bit longer (1 s. vs. 12 s.).

Benchs	Direct Cells		Configured Cells	
	Area	CPU	Area	CPU
c1355	826	170.3	822	172.3
c1908	1134	53.1	1130	54.0
c2670	1840	155.3	1831	155.6
c3540	2671	428.0	2651	431.3
c499	826	12.7	822	12.7
c5315	4216	325.0	4210	328.1
c6288	5486	32.1	5468	32.1
c7552	5264	368.4	5228	375.1
alu4	1675	404.6	1675	406.8
apex6	1580	164.6	1576	165.2
des	9542	1407.9	9511	1421.1
frg2	2913	268.7	2901	269.2
k2	4795	526.5	4789	529.9
pair	3664	568.3	3655	571.4
rot	1506	114.7	1500	115.0
too_large	1855	293.5	1832	294.5
vda	2325	187.3	2323	188.1
x3	2021	140.0	2010	140.9
TOTAL	54049	5621.0	53934	5663.5
GAIN	-	-	0.2%	+1.0x

Table 4: Results with Direct Cells and Configured Cells.

6 Conclusion and Future Work

An original method has been presented where we have shown that a single ROBDD construction of a given function f can be sufficient to solve the complete matching problem between f and a set of cells. *Implicit Pattern Matching* can be considered as a method defining a perfect key which allows to access all the matching cells at a time. The resolution is done by using the internal BDD unique table which is known to be very efficient. The key concept is to both construct BDDs of cell functionalities and BDDs of subject graphs (during the technology mapping) upon the same set of variables and not separately like in classical methods. Experimental results showed also that the implementation of this approach can bring important speed-up (even by inserting double inverters in the original Boolean network) and exhibited new optimum for a well known set of benchmarks. Because of the speed of this fundamental approach, we can easily embed it in a classical dynamic programming algorithm which exhaustively looks for the best solution. Moreover, we have presented an efficient way to represent cell functionalities through a single network of BDDs and by representing also naturally the configured cells.

In the future, a general improvement of the method must consist

to reduce the *Canonical representants Sets* of the cells functions - as we have done in this paper by introducing the notion of *compatible variable assignments* with a signature S - in order to handle bigger cells. Practically, cells with more than 10 inputs can lead to huge *Canonical representants Sets* and thus the Canonical representant Set of the library may not be able to be completely generated. These improvements must take into account the work done by [4] but without touching the main property of Implicit Pattern Matching which behaves like a perfect key.

References

- [1] L.Benini, G.De Micheli, "A survey of Boolean matching techniques for library binding", on <http://www.acm.org/todae/V2N3/L224/abstract.html>.
- [2] R.K Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: a Multiple-Level Logic Optimization System". in IEEE Transactions on Computer-Aided Design, CAD-6(6):1062-1081, 1987.
- [3] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. On Computer*, 35(8)-677-691, 1986
- [4] J.R. Burch, D.E. Long, "Efficient Boolean Function Matching", in *Proceedings of the International Conference on Computer-Aided-Design*, pages 408-411, 1992.
- [5] K.C. Chen, "Boolean matching based on Boolean unification", in *Proceedings of the Euro-Dac*, pages 346-351, 1993.
- [6] O.Coudert, "SIAM: Une boite à outil pour la preuve formelle de systèmes séquentiels", Thesis at *Ecole Nationale Supérieure des Télécommunications*, 1991.
- [7] S. Ercolani, G.De Micheli, "Technology mapping for electrically programmable gate arrays", in *Proceedings of Design Automation Conference*, pages 234-239, 1991.
- ✗ [8] U.Hinsberger, R.Kolla, "Boolean Matching for large libraries", In *Proceedings of Design Automation Conference*, pages 206-211, 1998.

- [9] K.Keutzer. "DAGON: Technology Binding and Local Optimization by DAG Matching". in *Proceedings of Design Automation Conference*, pages 341-347, 1987.
- [10] F. Mailhot, G.De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations", *IEEE Transactions on CAD/ICAS*, Vol. 12, No. 5, pages 599-620, 1993.
- [11] S. Minato, N. Ishiura, S. Yajima, "Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation", in Proceedings of 27th Design Automation Conference, 1990.
- X [12] U. Schlichtmann, F. Brélez, M. Hermann, "Characterization of Boolean Functions for Rapid Matching in EPLA Technology Mapping", in Proceedings of Design Automation Conference, pages 374-379, 1992.
- [13] K. Zhu, D.F. Wong, "Fast Boolean Matching for Field-Programmable Gate Arrays", in *Proceedings of the Euro-Dac*, pages 352-357, 1993.